Functional Netlists *

Sungwoo Park Jinha Kim Hyeonseung Im

Department of Computer Science and Engineering
Pohang University of Science and Technology
Gyeongbuk, Republic of Korea
{gla,goldbar,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 variant of the lambda calculus, called $l\lambda$ (linear lambda), which may serve as a high-level substitute for netlists. 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 high-level substitute for netlists, we design a simple hardware description language that extends $l\lambda$ with polymorphism, and use it to implement a Fast Fourier Transform circuit.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics, Syntax

General Terms Languages

Keywords Hardware description language, Functional language, Linear type system

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 and Rasmussen 1995; O'Donnell 1995; Bjesse et al. 1998; Matthews et al. 1998; Li and Leeser 2000; Mycroft and 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 and Plotkin 1981; Sheeran 1984; Johnson 1984; Boute 1984; Meshkinpour and Ercegovac 1985) which saw the birth of currently popular hardware description languages. The merits of functional 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 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 lambda 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 lambda 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 lambda calculus as a high-level substitute for netlists, or as *functional netlists.*

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 lambda calculus as functional netlists. For example, we can express various higher-order combinators within the lambda calculus itself without recourse to additional meta-programming constructs or another host language.

To correctly translate higher-order functions, we need to take into consideration the fact that hardware circuits are physical re-

* This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MOST) (No. R01-2007-000-11087-0).

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

sources that cannot be shared in general. Consider a function

$$k = \lambda x\!:\!\mathbf{1}.\ \texttt{and}\ x\ x$$

where $\mathbf{1}$ is a base type for bitstreams and `and` denotes an 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

$$g = \lambda f\!:\!\mathbf{1} \rightarrow \mathbf{1}.\ f\ (f\ 0).$$

Since f represents a hardware circuit that takes a bitstream to emit another bitstream and thus 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 a number 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 types are used exactly once, *i.e.*, linearly.

Building upon these observations, we design a variant of the lambda calculus, called $l\lambda$ (linear lambda), which uses a *linear type system* to enforce the linear use of variables of function type. 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 substitute for netlists.

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 circuit. The actual code for the circuit is 60 lines long and expands to 5158 lines of Verilog code by our prototype translator.

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	$::=$	$\cdot \mid \Delta, f : \kappa$

Figure 1. Abstract syntax of $l\lambda$

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.

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 a Fast Fourier Transform circuit implemented in $l\lambda$ extended with polymorphism. Section 7 discusses related work and Section 8 concludes.

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) and that fixed point expressions permit only sharable variables in their binders.

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

$$\frac{\Gamma, x : \theta; \Delta \vdash e : \tau}{\Gamma; \Delta \vdash \lambda x{:}\theta.\, e : \theta \to \tau}\ {\to}\mathsf{I} \qquad \frac{\Gamma; \Delta, f : \kappa \vdash e : \tau}{\Gamma; \Delta \vdash \hat{\lambda} f{:}\kappa.\, e : \kappa \multimap \tau}\ {\multimap}\mathsf{I}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : \theta \to \tau \quad \Gamma; \Delta_2 \vdash e_2 : \theta}{\Gamma; \Delta_1, \Delta_2 \vdash e_1\ e_2 : \tau}\ {\to}\mathsf{E}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : \kappa \multimap \tau \quad \Gamma; \Delta_2 \vdash e_2 : \kappa}{\Gamma; \Delta_1, \Delta_2 \vdash e_1\,\hat{}\,e_2 : \tau}\ {\multimap}\mathsf{E}$$

$$\frac{\Gamma; \Delta_1 \vdash e_1 : \theta_1 \quad \Gamma; \Delta_2 \vdash e_2 : \theta_2}{\Gamma; \Delta_1, \Delta_2 \vdash (e_1, e_2) : \theta_1 \times \theta_2}\ {\times}\mathsf{I}$$

$$\frac{\Gamma; \Delta \vdash e : \theta_1 \times \theta_2 \quad \Gamma, x_1 : \theta_1, x_2 : \theta_2; \Delta' \vdash e' : \tau}{\Gamma; \Delta, \Delta' \vdash \mathbf{proj}\ e\ \mathbf{of}\ (x_1, x_2)\ \mathbf{in}\ e' : \tau}\ {\times}\mathsf{E}$$

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

Figure 2. Type system of $l\lambda$

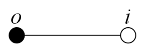
In order to simplify the presentation of the definition of $l\lambda$, we choose to syntactically distinguish between *sharable variables* x (of sharable type) and *linear variables* f (of linear type). Accordingly we use two kinds of functions and applications: $\lambda x{:}\theta.\, e$ and $e\ e$ for sharable input types and $\hat{\lambda} f{:}\kappa.\, e$ and $e\,\hat{}\,e$ for linear input types. Pairs (e, e) and projections $\mathbf{proj}\ e\ \mathbf{of}\ (x, x)\ \mathbf{in}\ e$ are expressions for product types. We use fixed point expressions $\mathbf{fix}\ x{:}\theta.\, e$ to build feedback circuits. 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 ${\to}\mathsf{E}$ and ${\multimap}\mathsf{E}$, 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 $\mathbf{1} \to \mathbf{1}$ because it emits a bitstream fed as input (after a delay). For `and`, we assign either $\mathbf{1} \to (\mathbf{1} \to \mathbf{1})$ or $(\mathbf{1} \times \mathbf{1}) \to \mathbf{1}$.

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.

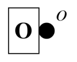
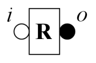
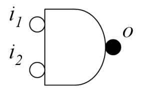
At the physical level, a hardware circuit consists of hardware components and connecting wires. A hardware component has one or more terminals to which external wires can be connected. We assume that every wire is unidirectional and never alternates the direction of the bitstream it transmits. Hence 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.*

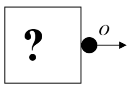
Proof. Suppose that $C_1 \cup C_2$ is not realizable: there is an input terminal i such that $o \mapsto i \in C_1 \cup C_2$, $o' \mapsto i \in C_1 \cup C_2$, and $o \neq o'$. Since $i \notin |C_1| \cap |C_2|$, we have either $o \mapsto i \in C_1$, $o' \mapsto i \in C_1$ (meaning that C_1 is not realizable) 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 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 as 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:

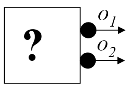
$$e_1 : \mathbf{1} \quad \Rightarrow$$



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:

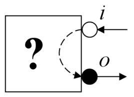
$$e2 : \mathbf{1} \times \mathbf{1} \quad \Rightarrow$$



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.

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:

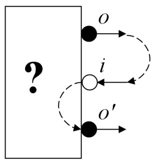
$$e_3 : \mathbf{1} \rightarrow \mathbf{1} \quad \Rightarrow$$



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':

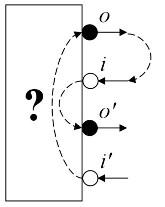
$$e_4 : (\mathbf{1} \rightarrow \mathbf{1}) \multimap \mathbf{1} \quad \Rightarrow$$



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':

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

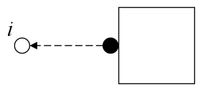


The external hardware circuit should be generated from an expression of type $\mathbf{1} \rightarrow \mathbf{1}$. For the output interface for e_5, we write $(o \rightarrow i) \multimap (i' \rightarrow o')$ to indicate that a bitstream flows from i' to o' and from o to i.

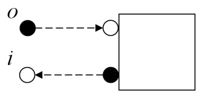
Examples 3, 4, and 5 illustrate that output interfaces for expressions of function type consist not only of output terminals but also of input terminals. For example, the output interface $i \rightarrow o$ for expression e_3 includes input terminal i to receive a bitstream from an external hardware circuit (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.

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

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

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 $|S| \cap |S'| = \varnothing$ holds where $|S|$ denotes the set of input and output terminals in S:

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

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 $o \in |I|$.

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

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

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

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

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

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

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

$O \bowtie I$ implicitly assumes that O and I are syntactically compatible (*e.g.*, $O_1 \times O_2 \bowtie O' \rightarrow I'$ never holds). Proposition 2.4 shows that an output interface and an input interface of the same type can be safely connected if both share no input terminal.

Proposition 2.4. *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 $l\lambda$ maps an expression to a tuple (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.

2.4 Connection points

In $l\lambda$, we can write expressions that describe not actual hardware circuits but 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.

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 tuple (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:

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

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.

Sharable input function

Consider an identify function $\lambda x{:}\mathbf{1}.\, x$ of type $\mathbf{1} \rightarrow \mathbf{1}$. 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 $i \rightarrow o$ because as a function of type $\mathbf{1} \rightarrow \mathbf{1}$, it receives a bitstream via i to emit another bitstream via o:

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

Note that $i \rightarrow o$ also has type $\mathbf{1} \rightarrow \mathbf{1}$, *i.e.*, $i \rightarrow o \rhd \mathbf{1} \rightarrow \mathbf{1}$.

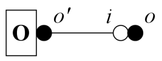
Note that it is not the instance of x in the body but the binder $x : \mathbf{1}$ that generates $\mathsf{pt}[i, o]$. For example, even if the body changes from x to (x, x), we do not generate an additional connection point. Instead we only update the output interface from $i \rightarrow o$ to $i \rightarrow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

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

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:

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

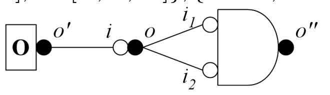


To bind the two instances of o in output interface (o, o) to different sharable variables, we use a projection. For example, the following expression binds the two instances of o to sharable variables y and z:

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

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

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

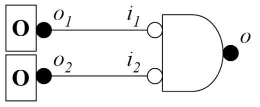


The above expression is equivalent to $(\lambda x\!:\!\mathbf{1}.\ \mathtt{and}\ x\ x)\ \mathtt{zero}$ which produces a hardware circuit with the same structure:

$$\begin{array}{l}(\lambda x\!:\!\mathbf{1}.\ \mathtt{and}\ x\ x)\ \mathtt{zero} \\ \quad\Rightarrow (\{\mathsf{pt}[i,o], \mathsf{0}[o'], \mathsf{and}[i_1,i_2,o'']\}, \{o' \mapsto i, o \mapsto i_1, o \mapsto i_2\}, o'')\end{array}$$

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

$$\begin{array}{l}\mathtt{and\ zero\ zero} \\ \quad\Rightarrow (\{\mathsf{0}[o_1], \mathsf{0}[o_2], \mathsf{and}[i_1,i_2,o]\}, \{o_1 \mapsto i_1, o_2 \mapsto i_2\}, o)\end{array}$$

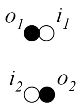


Linear input function

Consider another identify function $\hat\lambda f\!:\!\mathbf{1}\!\to\!\mathbf{1}.\ f$ of type $(\mathbf{1}\!\to\!\mathbf{1}) \multimap (\mathbf{1}\!\to\!\mathbf{1})$. 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 $\mathbf{1}\!\to\!\mathbf{1}$ consists of a pair of input and output terminals. Thus an input interface of type $\mathbf{1}\!\to\!\mathbf{1}$ consists of a pair of output and input terminals.

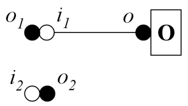
It is important that these output and input terminals for the input interface, say, o and i, must belong to separate connection points so that we can exploit an external hardware circuit providing an output interface of type $\mathbf{1}\!\to\!\mathbf{1}$ in the intended way, *i.e.*, by transmitting a 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 $o_1 \!\to\! i_2$ for its input interface and $i_1 \!\to\! o_2$ for its output interface. Then the output interface for $\hat\lambda f\!:\!\mathbf{1}\!\to\!\mathbf{1}.\ f$ becomes $(o_1 \!\to\! i_2) \multimap (i_1 \!\to\! o_2)$:

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



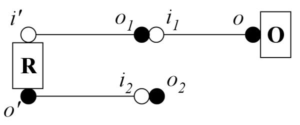
If the body changes from f to $f\ \mathtt{zero}$, we associate a new hardware component $\mathsf{0}[o]$ with $\mathtt{zero}$ and connect output terminal o to the input terminal in the output interface for f, namely i_1. The output interface changes to $(o_1 \!\to\! i_2) \multimap o_2$ because i_1 in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

$$\begin{array}{l}\hat\lambda f\!:\!\mathbf{1}\!\to\!\mathbf{1}.\ f\ \mathtt{zero} \\ \quad\Rightarrow (\{\mathsf{pt}[i_1,o_1], \mathsf{pt}[i_2,o_2], \mathsf{0}[o]\}, \{o \mapsto i_1\}, (o_1 \!\to\! i_2) \multimap o_2)\end{array}$$



Let us apply the resultant function to $\mathtt{reg}$. We associate a new hardware component $\mathsf{reg}[i', o']$ with $\mathtt{reg}$, and introduce two connection constraints so that the output interface $i' \!\to\! o'$ for $\mathtt{reg}$ 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:

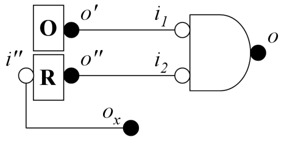
$$\begin{array}{l}(\hat\lambda f\!:\!\mathbf{1}\!\to\!\mathbf{1}.\ f\ \mathtt{zero})\hat{}\,\mathtt{reg} \\ \quad\Rightarrow (\{\mathsf{pt}[i_1,o_1], \mathsf{pt}[i_2,o_2], \mathsf{0}[o], \mathsf{reg}[i',o']\}, \\ \qquad\qquad \{o \mapsto i_1, o_1 \mapsto i', o' \mapsto i_2\}, o_2)\end{array}$$



Fixed point expression

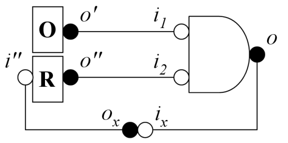
A fixed point expression $\mathbf{fix}\ x\!:\!\theta.\ e$ builds a feedback circuit whose output is accessible to itself via sharable variable x. As an example, let us build a feedback circuit from $\mathbf{fix}\ x\!:\!\mathbf{1}.\ \mathtt{and\ zero}\ (\mathtt{reg}\ x)$. We associate hardware components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

$$\begin{array}{l}\mathtt{and\ zero}\ (\mathtt{reg}\ x) \\ \quad\Rightarrow (\{\mathsf{and}[i_1,i_2,o], \mathsf{0}[o'], \mathsf{reg}[i'',o'']\}, \\ \qquad\qquad \{o' \mapsto i_1, o'' \mapsto i_2, o_x \mapsto i''\}, o)\end{array}$$



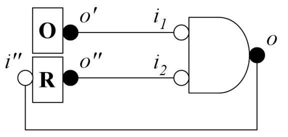
Now there are two ways to complete the feedback circuit, depending on whether we generate a connection point for x or not. First we associate an actual connection point $\mathsf{pt}[i_x, o_x]$ with x and connect o to i_x:

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



Second we identify output terminal o_x with output interface o for the whole hardware circuit, *i.e.*, by enforcing $o_x = o$:

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



The two hardware circuits are equivalent because a connection point does not occupy a physical area. The translation of $l\lambda$ uses the first approach which does not have to deal with equations between output terminals such as $o_x = o$.

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 the hardware synthesis process (for rewriting

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

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

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

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

Figure 4. Rules for translating types

an expression by analyzing its behavior so that it can be mapped directly to a hardware circuit), which is beyond the scope of this paper.

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.*, $|S| \cap |S'| = \varnothing$, 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* $O \rhd \tau$.

3.3 Translation of expressions

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

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

A binding $x :: O$ in $\mathcal{G}$ 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 $\mathcal{D}$ means that O is the output interface for variable f, and we must use f exactly once.

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

- We write $\mathcal{G} \sim \Gamma$ to mean that $x :: O \in \mathcal{G}$ 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 $\mathcal{D} \sim \Delta$ to mean that $f :: O \in \mathcal{D}$ 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, $f_1 :: O_1 \in \mathcal{D}$ and $f_2 :: O_2 \in \mathcal{D}$ with $f_1 \neq f_2$ mean $\sharp\{O_1, O_2\}$.

Note that while output interfaces in $\mathcal{D}$ do not share terminals, output interfaces in $\mathcal{G}$ may share terminals. Then variables declared in projections (*e.g.*, x and y in $\mathbf{proj}\ e\ \mathbf{of}\ (x, y)\ \mathbf{in}\ e'$) 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 $\mathcal{G}; \mathcal{D} \vdash e \Rightarrow (H, C, O)$. 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 $|S| \cap |S'| = \varnothing$. We calculate $|\mathcal{G}|$ and $|\mathcal{D}|$ as follows:

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

For S and S' in $\sharp\{S, S'\}$, we allow unions of different kinds of sets written as $\mathcal{G} + \mathcal{D}$, $\mathcal{D} + H$, $\mathcal{G} + \mathcal{D} + H$, 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 $|\mathcal{G} + \mathcal{D} + H| = |\mathcal{G}| \cup |\mathcal{D}| \cup |H|$.

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 $\to I$, $\multimap I$, and *Fix*.
- Connection constraints are generated only by the rules $\to E$, $\multimap E$, and *Fix*.
- In the rule $\multimap I$, $\mathcal{D}, f :: O_f \sim \Delta, f : \kappa$ holds from $\mathcal{D} \sim \Delta$, $O_f \rhd \kappa$, and $\sharp\{O_f, \mathcal{D}\}$.
- In the rules $\to I$ and *Fix*, $\sharp\{I_x, \mathcal{D} + H\}$ implies $\sharp\{I_x, \mathcal{G} + \mathcal{D} + H\}$ because $I_x \lhd \theta$ holds by Lemma 3.1, $|I_x|$ contains no output terminals by Proposition 2.3, and $|\mathcal{G}|$ contains only output terminals by Proposition 2.3.
- 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 $\mathcal{G}$. It explains why output interfaces in a sharable output context may share terminals.
- Because of sharable variables declared in projections, $\mathcal{G}; \mathcal{D} \vdash e \Rightarrow (H, C, O)$ may not satisfy $\sharp\{\mathcal{G}, \mathcal{D}\}$. That is, $\mathcal{G}$ and $\mathcal{D}$ may not be completely disjoint. For example, assuming $f :: I \to (O_1, O_2)$, a projection $\mathbf{proj}\ f\ e'\ \mathbf{of}\ (x_1, x_2)\ \mathbf{in}\ e$ 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 $\mathbf{1} \to (\mathbf{1} \to \mathbf{1})$, we may use the following rule:

$$\frac{\{i_1, i_2, o\} \not\subset |\mathcal{G}| \quad i_1 \neq i_2}{\mathcal{G}; \cdot \vdash \mathsf{and} \Rightarrow (\{\mathsf{and}[i_1, i_2, o]\}, \cdot, i_1 \to (i_2 \to o))}\ \mathit{And}$$

Although we may read $\mathcal{G}; \mathcal{D} \vdash e \Rightarrow (H, C, O)$ operationally by regarding $\mathcal{G}$, $\mathcal{D}$, 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. We can

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

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

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

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

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

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

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

$$\frac{\theta \lhd\!\rhd (H_x, I_x, O_x) \quad \mathcal{G}, x :: O_x; \mathcal{D} \vdash e \Rightarrow (H, C, O) \quad \sharp\{O_x, \mathcal{G} + \mathcal{D}\} \quad \sharp\{I_x, \mathcal{D} + H\}}{\mathcal{G}; \mathcal{D} \vdash \mathbf{fix}\ x : \theta.\, e \Rightarrow (H_x \cup H, C \cup O \bowtie I_x, O)}\ \mathit{Fix}$$

Figure 5. Rules for translating expressions

make the translation more declarative by rewriting the rule Fix so that it does not have to translate θ to create new connection points:

$$\frac{\mathcal{G}, x :: O; \mathcal{D} \vdash e \Rightarrow (H, C, O) \quad \sharp\{O, \mathcal{G} + \mathcal{D}\}}{\mathcal{G}; \mathcal{D} \vdash \mathbf{fix}\ x : \theta.\, e \Rightarrow (H, C, O)}\ \mathit{Fix}'$$

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 (Section 4.3).

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 4.6).
- 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* $\cdot; \cdot \vdash e : \tau$*,* $O \rhd \tau$*, 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 4.3 and 4.5.

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

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

Proposition 4.3. *If* $\mathcal{G}; \mathcal{D} \vdash e \Rightarrow (H, C, O)$ *with* $\mathcal{G} \sim \Gamma$ *and* $\mathcal{D} \sim \Delta$*, then* $\Gamma; \Delta \vdash e : \tau$ *and* $O \rhd \tau$*.*

Proof. By induction on the structure of the proof of $\mathcal{G}; \mathcal{D} \vdash e \Rightarrow (H, C, O)$. The case for the rule $\times I$ uses the premises $O_1 \rhd \theta_1$ and $O_2 \rhd \theta_2$. ☐

Lemma 4.4. *If* $\mathcal{G}; \mathcal{D} \vdash e \Rightarrow (H, C, O)$ *with* $\mathcal{G} \sim \Gamma$ *and* $\mathcal{D} \sim \Delta$*, then if* $i \in |O|$*, then* $o \mapsto i \notin C$*.*

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

Proposition 4.5. *If* $\mathcal{G}; \mathcal{D} \vdash e \Rightarrow (H, C, O)$ *with* $\mathcal{G} \sim \Gamma$ *and* $\mathcal{D} \sim \Delta$*, then* C *is realizable.*

Proof. By induction on the structure of the proof of $\mathcal{G}; \mathcal{D} \vdash e \Rightarrow (H, C, O)$. The proof reuses the result from the proof of Proposition 4.3 that all output contexts are well-formed. ☐

4.2 Soundness of the type system

Theorem 4.6. *If* $\cdot; \cdot \vdash e : \tau$*, then there exists* (H, C, O) *such that* $\cdot; \cdot \vdash e \Rightarrow (H, C, O)$*.*

Theorem 4.6 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 4.8.

Lemma 4.7. *If* $\mathcal{G} \sim \Gamma$*,* $\mathcal{D} \sim \Delta$*, and* $\mathcal{G}; \mathcal{D} \vdash e \Rightarrow (H, C, O)$*, then* $\sharp\{\mathcal{G} + \mathcal{D}, H\}$*.*

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

Proposition 4.8. *If $\Gamma; \Delta \vdash e : \tau$, then for $\mathcal{G} \sim \Gamma$ and $\mathcal{D} \sim \Delta$, there exists (H, C, O) such that $\mathcal{G}; \mathcal{D} \vdash e \Rightarrow (H, C, O)$ and $O \rhd \tau$.*

Since the translation uses a declarative style, a strict proof of Proposition 4.8 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 $\mathcal{G}; \mathcal{D} \vdash e \Rightarrow (H, C, O)$ 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 4.8 assumes that the last two premises in each of the rules $\to I$, $\multimap I$, and Fix automatically hold.

For $\mathcal{G}; \mathcal{D} \vdash e \Rightarrow (H, C, O)$, we take $\mathcal{G}$, $\mathcal{D}$, and e as input and (H, C, O) as output. Since H shares no terminals with $\mathcal{G}$ and $\mathcal{D}$ by Lemma 4.7, we further assume that all terminals in H are assigned fresh identifiers. That is, given $\mathcal{G}; \mathcal{D} \vdash e \Rightarrow (H, C, O)$, 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 4.8 assumes that the last premise in each of the rules $\to E$, $\multimap E$, $\times I$, $\times E$ automatically holds.

Proof of Proposition 4.8. By induction on the structure of the proof of $\Gamma; \Delta \vdash e : \tau$. ☐

4.3 Completeness of the translation

In order for $l\lambda$ to be a substitute for netlists, 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 show that $l\lambda$ is indeed expressive enough to describe every realizable hardware circuit. We assume tuple types $\theta_1 \times \cdots \times \theta_n$ generalizing product types, tuples $(e_1, \cdots, e_n)$ generalizing pairs, tuple patterns $(p_1, \cdots, p_n)$ generalizing pair patterns, and allow tuple patterns in fixed point expressions and projections (*e.g.*, $\mathbf{fix}\ (p_1, \cdots, p_n) : \theta.\ e$ and $\mathbf{proj}\ e\ \mathbf{of}\ (p_1, \cdots, p_n)\ \mathbf{in}\ e'$).

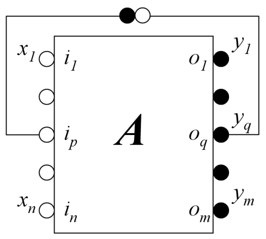
Consider a hardware circuit $\mathcal{A}$ with n input terminals $i_1, \cdots, i_n$ and m output terminals $o_1, \cdots, o_m$. Here we enumerate all output terminals belonging to $\mathcal{A}$, but exclude those "hidden" input terminals to which wires are already connected. That is, we consider only those input terminals exposed to external hardware circuits. We assume that $\mathcal{A}$ 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 $(1 \le p \le n)$ and e has type $\mathbf{1} \times \cdots \times \mathbf{1}$ whose q-th element corresponds to output terminal o_q $(1 \le q \le m)$.

We observe that there are two ways to augment $\mathcal{A}$. 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 $y_1, \cdots, y_{q-1}, y_{q+1}, \cdots, y_m$:

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



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

$$\lambda x'_1 : \mathbf{1}. \cdots \lambda x'_l : \mathbf{1}.\ e'$$

where x'_r corresponds to its r-th input terminal $(1 \le r \le l)$ and e' produces k output terminals. Then $\mathcal{A}$ combined with $\mathcal{A}'$ is described by the following expression:

$$\begin{array}{l} \lambda x_1 : \mathbf{1}. \cdots \lambda x_n : \mathbf{1}.\ \lambda x'_1 : \mathbf{1}. \cdots \lambda x'_l : \mathbf{1}. \\ \quad \mathbf{proj}\ e\ \mathbf{of}\ (y_1, \cdots, y_m)\ \mathbf{in} \\ \qquad \mathbf{proj}\ e'\ \mathbf{of}\ (y'_1, \cdots, y'_k)\ \mathbf{in}\ (y_1, \cdots, y_m, y'_1, \cdots, y'_k) \end{array}$$

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 $\mathcal{A}$.

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 $\lambda x_1 : \mathbf{1}.\ \lambda x_2 : \mathbf{1}.$ `and` $x_1\ x_2$).

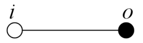
5. Discussion

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.

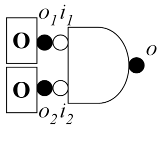
An alternative translation of $l\lambda$ dispenses with connection points and maps variables directly to wires. The idea is to treat wires as independent hardware components with their own input and output terminals. We can obtain such a translation by reusing the previous translation of $l\lambda$ with 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:

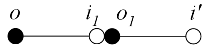
$$\begin{array}{l} \texttt{and zero zero} \\ \quad \Rightarrow (\{\mathsf{0}[o_1], \mathsf{0}[o_2], \mathsf{and}[i_1, i_2, o]\}, \{o_1 \mapsto i_1, o_2 \mapsto i_2\}, o) \end{array}$$



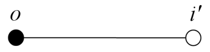
If we again choose to realize connection constraints as wires, it suffices to interpret $\mathsf{pt}[i, o]$ as a wire of zero length, *i.e.*, as a connection point. Then we obtain the original translation of $l\lambda$ given in Section 3.

5.2 Eliminating redundant wires

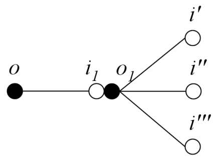
The translation of $l\lambda$ ensures that well-typed expressions are always mapped to realizable hardware circuits, but it sometimes produces redundant wires if all connection constraints are realized as wires. For example, $(\hat{\lambda} f : \mathbf{1} \to \mathbf{1}.\, f\ \mathtt{zero})\,\hat{}\,\mathtt{reg}$ 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:



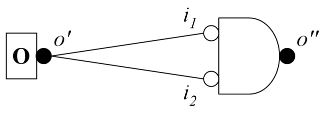
If we wish to eliminate such redundant wires, we can treat input terminals of connection points in the following way. We write $\overline{o}$ for the input terminal of a connection point whose output terminal is o. Now every connection point is written as $\mathsf{pt}[\overline{o}, o]$:

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

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 does not produces an otherwise redundant wire.

As an example, $(\lambda x : \mathbf{1}.\, \mathtt{and}\ x\ x)\ \mathtt{zero}$ in Section 3.1 now produces a hardware circuit with no redundant wire:

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



6. Extension of $l\lambda$

As a high-level substitute for netlists, $l\lambda$ 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 non-trivial hardware circuits in a concise way. This section presents an extension of $l\lambda$ with polymorphism and an implementation of a Fast Fourier Transform (FFT) circuit.

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.

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

$$\frac{\Gamma; \Delta \vdash e : \sigma \quad \alpha \notin tvar(\Gamma \cup \Delta)}{\Gamma; \Delta \vdash \Lambda\alpha.\, e : \forall\alpha.\sigma}\ \forall I$$

$$\frac{\Gamma; \Delta \vdash e : \forall\alpha.\sigma}{\Gamma; \Delta \vdash e\,\langle\theta\rangle : [\theta/\alpha]\sigma}\ \forall E$$

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 $l\lambda$. 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 Example - Fast Fourier Transform

Our implementation of the FFT circuit 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:

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

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:

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

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

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

Figure 6. Fast Fourier Transform in $l\lambda$

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'. $e_1\circ e_2$ 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.

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 complex numbers and three constants `cplus`, `cminus`, and `cmult`, all of type $\mathtt{c}^{2}\to\mathtt{c}$, 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 $\alpha\to\alpha$ 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 additional functions for reordering the output) and expands to 5158 lines of Verilog code by our prototype translator. We have also implemented a bitonic sorting network whose code is 43 lines long and expands to 5175 lines of Verilog code. The generated Verilog code has been tested for correctness on Aldec's Active-HDL simulator.

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 pre-calculated 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 more 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 and Leeser 2000) is embedded into ML. An example of a functional language designed 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 datatype representing hardware circuits; Claessen and 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). Their use enables 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, therefore, 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 and 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 and 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 and 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 variant of the lambda calculus, called $l\lambda$, which may serve as a high-level substitute for netlists. 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 high-level substitute for netlists by implementing a Fast Fourier Transform circuit.

Although $l\lambda$ is designed primarily as a high-level substitute for netlists, developing it into a 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 $l\lambda$ to express such physical properties of hardware circuits as layout and wiring. Combined together, these two directions will turn $l\lambda$ into a practical functional hardware description language.

Acknowledgments

The authors are grateful to anonymous reviewers for their helpful comments.

References

Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired: wire-aware circuit design. In *Proceedings of Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 3725 of *Lecture Notes in Computer Science*. Springer Verlag, 2005.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in Haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 174–184. ACM Press, 1998.

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

Luca Cardelli and Gordon Plotkin. An algebraic approach to VLSI design. In *Proceedings of the VLSI 81 International Conference*, pages 173–182, 1981.

Koen Claessen and David Sands. Observable sharing for functional circuit description. In *ASIAN '99: Proceedings of the 5th Asian Computing Science Conference on Advances in Computing Science*, pages 62–73. Springer-Verlag, 1999.

Dan R. Ghica. Geometry of synthesis: a structured approach to VLSI design. In *Proceedings of the 34th annual ACM*

SIGPLAN-SIGACT symposium on Principles of programming languages, pages 363–375. ACM Press, 2007.

Jim Grundy, Tom Melham, and John O'Leary. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, 16(2):157–196, 2006.

Steven D. Johnson. Applicative programming and digital design. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 218–227. ACM Press, 1984.

Geraint Jones and Mary Sheeran. Circuit design in Ruby. *Formal Methods in VLSI Design*, pages 13–70, 1990.

Yanbing Li and Miriam Leeser. HML, a novel hardware description language and its translation to VHDL. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(1):1–8, 2000.

John Matthews, Byron Cook, and John Launchbury. Microprocessor specification in Hawk. In *Proceedings of the 1998 International Conference on Computer Languages*, page 90. IEEE, 1998.

F. Meshkinpour and Milos D. Ercegovac. A functional language for description and design of digital systems: sequential constructs. In *Proceedings of the 22nd ACM/IEEE conference on Design automation*, pages 238–244. ACM press, 1985.

Alan Mycroft and Richard Sharp. A statically allocated parallel functional language. In *ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 37–48. Springer-Verlag, 2000.

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

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

Peter O'Hearn. On bunched typing. *Journal of Functional Programming*, 13(4):747–796, 2003.

Robin Sharp and Ole Rasmussen. Using a language of functions and relations for VLSI specification. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 45–54. ACM Press, 1995.

Mary Sheeran. Hardware design and functional programming: a perfect match. *The Journal of Universal Computer Science*, 11 (7):1135–1158, 2005.

Mary Sheeran. muFP, a language for VLSI design. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 104–112. ACM Press, 1984.