A Formalism for Higher-Order Composition Languages that Satisfies the Church-Rosser Property

Adam Cataldo
Elaine Cheong
Thomas Huining Feng
Edward A. Lee
Andrew Christopher Mihal

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-48
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-48.html

May 9, 2006

Acknowledgement

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF award #CCR-0225610), the State of California Micro Program, and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft, and Toyota.

A Formalism for Higher-Order Composition Languages that Satisfies the Church-Rosser Property

Adam Cataldo Elaine Cheong Thomas Huining Feng
Edward A. Lee Andrew Mihal

Abstract

In actor-oriented design, programmers make hierarchical compositions of concurrent components. As embedded systems become increasingly complex, these compositions become correspondingly complex in the number of actors, the depth of hierarchies, and the connections between ports. We propose *higher-order composition languages* as a way to specify these actor-oriented models. The key to these languages is the ability to succinctly specify configurations with *higher-order parameters*—parameters that themselves might be configurations. We present a formalism which allows us to describe arbitrarily complex configurations of components with higher-order parameters. This formalism is an extension of the standard λ calculus.

1 Introduction

Actor-oriented design [19] is commonplace in embedded systems. In actor-oriented design, programmers model concurrent components (called *actors*) which communicate with one another through ports. *Configurations*, or hierarchies of interconnected components, are used to bundle networks of components into single components. Of course, each component may itself be thought of as a configuration, so we will cease to make a distinction between the two. The resulting systems are typically easier to reason about than those programmed with threads [20], since concurrent interaction is much more explicit. Examples of actor-oriented languages include hardware description languages [28], coordination languages [27], architecture description languages [23], synchronous languages [4], Giotto [14], SystemC [1], SHIM [12], CAL [13], and many more. Many software tools, including Simulink [10], LabVIEW [17], and Ptolemy II [7] provide graphical environments for actor-oriented design.

For large embedded systems, it may be appropriate to use different models of computation for component interaction at different levels of a model's hierarchy. These are called *heterogeneous* or *multi-paradigm* [31]. Systems metamodeling [26] tools, such as GME [18], make it easier to create domain specific modeling

environments, while model-based design tools, such as Ptolemy II, make it easier to construct models with heterogeneous behavior. Frameworks such as the tagged signal model [21, 5] make it easier to reason about such systems.

These approaches make it simpler to design systems with complex *semantics*. As the number of software and hardware components in embedded systems grows larger and larger, however, new techniques for simple *syntactic* descriptions of systems, whether visual or textual, will become equally important. For example, imagine designing a system with $10,000$ components. If these components must be instantiated and connected manually, the design process will be slow and error-prone. We wish to have automated tools with a strong mathematical basis to create such complex systems.

We thus propose *higher-order composition languages* as a way to specify actor-oriented models. At Berkeley, we are developing such a language that we call Ptalon. The key to these languages is the ability to succinctly specify configurations with *higher-order parameters*—parameters that themselves might be configurations. As an example, a parameterized configuration may describe a distributed sort application, with a "divide" component parameter, a "conquer" component parameter, and a parameter for the respective numbers of each component. A programmer will specify this configuration once and can then use it for an arbitrary number of components with arbitrary divide and conquer instances. This particular example is similar to the MapReduce programming pattern used by Google for distributed computation [11].

In fact, many methods of describing such systems have already been used in embedded system design. For instance, recursive structures can be specified in VHDL [22] to simplify hardware descriptions, and Bluespec System Verilog [6] brings such features to the system level through many forms of parameterization. Both [29] and [9] propose higher-order languages for embedded system design, and [16] suggests higher-order Petri nets, in which Petri nets themselves may be tokens, as a method to make more reusable Petri net structures. The Liberty Simulation Environment [30] supports higher-order parameters in its system modeling environment. π calculus [25] is widely used in describing connected and communicating components. However, π calculus as a mathematical basis lacks the Church-Rosser property [3]. Its non-determinism also makes it hard to reason deterministically about the design.

In the Ptalon project, we seek to take such work a step further by developing a simple, generic formalism in which we can describe configurations of components with higher-order parameters. We want the formalism to easily reflect structure found in current and future actor-oriented languages.

In any composition language, one should be able to:

1. Describe the interface of a component.
2. Describe the parallel composition of multiple components.
3. Establish links between components.
4. Create new components through hierarchy.

In addition to these four properties, in a higher-order composition language, one should also be able to:

5. Describe higher-order parameters to a configuration.

One of the first questions that comes up is, "Do we need types?" Standard types like characters and doubles seem largely irrelevant to a higher-order composition language. They certainly have uses, such as specifying the data types we wish to allow on particular ports, but we seek to focus on the larger problem of how to specify extremely flexible configurations via higher-order parameters.

If our formalism omits types, it may at first seem impossible to parameterize configurations with integers, such as a configuration which has a numeric parameter to specify how many instances of another configuration to wire together in series. In our formalism, representing a number is no different than representing a configuration, however, we have no such problem. This is the basic idea behind Church numerals in λ calculus, where numbers are represented by terms in the calculus. In this paper, we will show that the connection to λ calculus turns out to be much deeper than just that.

In particular, we will show how a slight extension of λ calculus provides a formalism for higher-order composition languages, free of any type system. We are not suggesting this as a concrete higher-order composition language but as a mathematical framework for composition languages in general. We are currently developing such a language, Ptalon, which will provide a more user-friendly syntax, but will be firmly grounded in the formalism given here, and will leverage its higher-order nature.

There are several practical insights that come out of the study of this theoretical model. Because our system is completely untyped, all the operators we create are *generic*; they can operate on data of any type. Because we extend λ calculus, higher-order functions are first class citizens of the calculus. Thus, many standard computational entities, such as numbers, boolean values, and ordered pairs, are simply particular higher-order functions in our calculus. Moreover, we will show that any configuration is also just another higher-order function. Finally, we will provide a justification that this syntax supports the kinds of systems we are interested in modeling.

We will try to keep the exposition self-contained, but we assume the reader has at least a basic understanding of λ calculus. To the novice, we recommend [3] as a great help in understanding the much larger context of λ calculus.

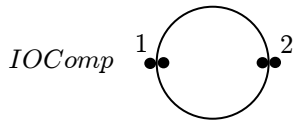
2 Some Examples in the Calculus

Before we rigidly define our calculus, we begin with a few example expressions. Consider a simple component which has two ports, named *in* and *out*. In our calculus, we might model this as

$$IOComp \equiv \lambda in.\lambda out.(\varepsilon\ (\oplus\ in\ out))$$

Here $\equiv$ means syntactic identity. Thus we use *IOComp* as a shorthand for the expression on the right. The $(\oplus\ in\ out)$ subexpression means to compose entities named *in* and *out* in parallel. The ε is an operator that in our calculus yields an encapsulation, or a hiding of arguments that are not explicitly exposed. The λin means to link the entity *in* to the interface, and the λout means to link the entity *out* to the interface.

We show a pictorial interpretation of this subexpression:



This picture is similar in both character and spirit of Milner's flowgraphs [24]. Note, however, that names are irrelevant; we number the ports to denote their corresponding order in the λ expression, but the following expression would have an identical picture:

$$IOComp' \equiv \lambda a.\lambda b.(\varepsilon\ (\oplus\ a\ b))$$

We can easily prove that for all x, y:

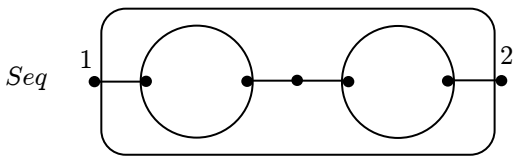
$$((IOComp\ x)\ y) = ((IOComp'\ x)\ y)$$

where $A = B$ means that A and B reduce to the same expression.

Now, suppose we wish to connect two *IOComp* components in series. There are actually several ways we can do it in this calculus, but we show one:

$$Seq \equiv \lambda in.\lambda out.(\varepsilon\ \lambda r.(\oplus\ ((IOComp\ in)\ r)\ ((IOComp\ r)\ out)))$$

In this expression, the ε encapsulates the relation r between the output of one *IOComp* component and the input of the other. r is no longer part of the interface. This expression will reduce to the expression with the picture below, with the dot in the middle connection representing this relation:



Now suppose we wish to connect an arbitrary number of *IOComp* components in series. We first encode numbers as higher-order models, using Church numerals [3]:

$$\begin{aligned} C_0 &\equiv Zero \\ C_{n+1} &\equiv Succ\ C_n \end{aligned}$$

where

$$\begin{aligned} Zero &\equiv \lambda f.\lambda x.x \\ Succ &\equiv \lambda c.\lambda f.\lambda x.(f\ ((c\ f)\ x)) \end{aligned}$$

We then define the following term:

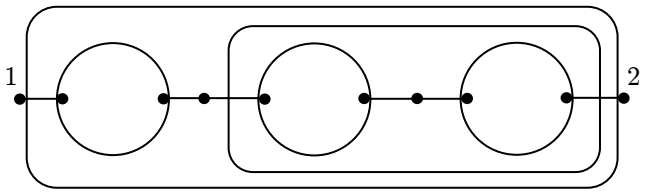
$$Series \equiv \lambda a.\lambda b.\lambda in.\lambda out.(\varepsilon\ \lambda r.(\oplus\ ((a\ in)\ r)\ ((b\ r)\ out)))$$

Then we can show that

$$((C_n\ (Series\ IOComp))\ IOComp) \tag{1}$$

represents $n+1$ copies of the $IOComp$ component linked in series. Specifically, when $n = 1$, (1) reduces to Seq; when $n = 2$, (1) reduces to the expression with the picture below:

$((C_2\ (Series\ IOComp))\ IOComp)$



Before we give a similar definition for parallel composition, we consider a few standard λ calculus terms:

$$True \equiv \lambda x.\lambda y.x \tag{2}$$
$$False \equiv \lambda x.\lambda y.y \tag{3}$$

The $\oplus$ operator that we have used thus far is our parallel composition operator. With the parallel reduction that we define later, we have

$$((\oplus\ A\ B)\ C) = (\oplus\ ((C\ True)\ A)\ ((C\ False)\ B))$$

This is a mechanism for passing parameters to the members of a composition. We now give the following nonstandard λ terms:

$$Left \equiv \lambda d.\lambda l.\lambda t.((l\ (t\ d))\ t) \tag{4}$$
$$Right \equiv \lambda d.\lambda l.\lambda t.((l\ t)\ (t\ d)) \tag{5}$$

This way if we compose A and B in parallel and wish to pass D to A, we can prove that

$$((\oplus\ A\ B)\ (Left\ D)) = (\oplus\ (A\ D)\ B)$$

Similarly if we wish to pass D to B, we can prove that

$$((\oplus\ A\ B)\ (Right\ D)) = (\oplus\ A\ (B\ D))$$

Now we define

$$Parallel \quad \equiv \quad \lambda a.\lambda b.(\oplus\ a\ b)$$

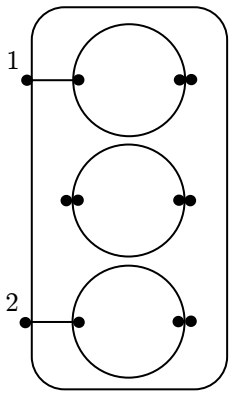
$((C_n\ (Parallel\ IOComp))\ IOComp)$ represents $n+1$ $IOComp$ components in parallel. As an example,

$$((C_2\ (Parallel\ IOComp))\ IOComp) =$$
$$(\oplus\ IOComp\ (\oplus\ IOComp\ IOComp))$$

This represents three components in parallel. If we wish to expose only the first port of the first $IOComp$ and the first port of the last component, we could write,

$\lambda a.\lambda b.(\varepsilon\ (((C_2\ (Parallel\ IOComp))\ IOComp)\ (Left\ a)\ (Right\ (Right\ b)))) =$
$\lambda a.\lambda b.(\varepsilon\ (\oplus\ (IOComp\ a)\ (\oplus\ IOComp\ (IOComp\ b))))$

This expression can be represented by the following picture.



3 The Formalism

We have now explained the basic building blocks of our calculus by example. In this section, we shall develop the formalism. Let V be a well-ordered, denumerable (countable and infinite) set of *variables*. By convention, x, y, z are members of V. We build a set $\mathcal{T}$ of *model terms* inductively, with $A, B, C \in \mathcal{T}$ by convention.

Definition 1. *The set $\mathcal{T}$ is defined inductively by:*

$$x \in V \quad \Rightarrow \quad x \in \mathcal{T} \tag{6}$$
$$A \in \mathcal{T} \quad \Rightarrow \quad (\varepsilon\ A) \in \mathcal{T} \tag{7}$$
$$A, B \in \mathcal{T} \quad \Rightarrow \quad (A\ B) \in \mathcal{T} \tag{8}$$
$$(\oplus\ A\ B) \in \mathcal{T} \tag{9}$$
$$A \in \mathcal{T}, x \in V \quad \Rightarrow \quad \lambda x.A \in \mathcal{T} \tag{10}$$

Rules 6, 8, and 10 by themselves define the syntax of *pure* λ *calculus* [3].

We will often omit parentheses. We note that application (Rule 8) is left associative, whereas abstraction (Rule 10) is right associative. Thus,

$$\lambda x.x\ z\ \lambda y.x\ y$$

is equivalent to

$$\lambda x.((x\ z)\ (\lambda y.(x\ y)))$$

We may also arbitrarily add parentheses to disambiguate terms.

We now define subterms. This extends the definition of subterms in [2].

Definition 2. *The* subterm function $\mathrm{Sub} : \mathcal{T} \to \mathcal{P}(\mathcal{T})$ *returns the set of subterms of each term argument:*

$$\begin{aligned}
\mathrm{Sub}(x) &= \{x\} \\
\mathrm{Sub}((\varepsilon\ A)) &= \mathrm{Sub}(A) \cup \{(\varepsilon\ A)\} \\
\mathrm{Sub}((A\ B)) &= \mathrm{Sub}(A) \cup \mathrm{Sub}(B) \cup \{(A\ B)\} \\
\mathrm{Sub}((\oplus\ A\ B)) &= \mathrm{Sub}(A) \cup \mathrm{Sub}(B) \cup \{(\oplus\ A\ B)\} \\
\mathrm{Sub}(\lambda x.A) &= \mathrm{Sub}(A) \cup \{\lambda x.A\}
\end{aligned}$$

If $A \in \mathrm{Sub}(B)$*, we write* $A \preccurlyeq B$*, and call* $\preccurlyeq$ *the* subterm order *on* $\mathcal{T}$*.*

We note that $(\mathcal{T}, \preccurlyeq)$ is a well-founded partial order, whose minimal elements are exactly the elements of V. A subterm A may *occur* in B several times. For instance, $\lambda x.x$ occurs in $(\oplus\ (\lambda x.x)\ (\lambda x.x))$ twice.

Definition 3. *The function* $\mathrm{FV} : \mathcal{T} \to \mathcal{P}(V)$*, which returns the set of* free variables $FV(A)$ *of a term* A*, is defined inductively as follows:*

$$\begin{aligned}
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}((\varepsilon\ A)) &= \mathrm{FV}(A) \\
\mathrm{FV}((A\ B)) &= \mathrm{FV}(A) \cup \mathrm{FV}(B) \\
\mathrm{FV}((\oplus\ A\ B)) &= \mathrm{FV}(A) \cup \mathrm{FV}(B) \\
\mathrm{FV}(\lambda x.A) &= \mathrm{FV}(A) \setminus \{x\}
\end{aligned}$$

As in [15], for subterm $\lambda x.A$ of B, A is called the *scope* of λx, and x is *bound* in A. Note that a bound variable is not free. We call a term B *closed* if $\mathrm{FV}(B) = \emptyset$.

Definition 4. *A term* A *is a* configuration *if and only if*

1. *It is a closed term. That is* $\mathrm{FV}(A) = \emptyset$*.*
2. *There do not exist terms* $B, C \in \mathcal{T}$ *such that* $(B\ C) \preccurlyeq A$*.*

Note that since all variables are bound in a term which defines a configuration, all variables in such terms correspond to links between entities or links to the interface. Any occurrence of $(\oplus\ A\ B)$ as a subterm corresponds to the parallel composition of entities A and B. Finally, any occurrence of the ε operator gives us hierarchy through encapsulation. While we exclude function application, a term with function application may reduce to a configuration, as the examples in the previous section demonstrated. Thus, a configuration in this form is "unparameterized."

Given a term A, we let $A[x := B]$ denote the *substitution* of all free occurrences of x in A with the term B. For example,

$$(x\ y\ (\lambda x.x))[x := (\lambda z.z)] = (\lambda z.z)\ y\ (\lambda x.x)$$

Formally,

$$\begin{aligned}
x[y := B] &\equiv \begin{cases} B & x \equiv y \\ x & x \not\equiv y \end{cases} \\
(\varepsilon\ A)\,[y := B] &\equiv (\varepsilon\ A[y := B]) \\
(A_1\ A_2)\,[y := B] &\equiv ((A_1[y := B])\ (A_2[y := B])) \\
(\oplus\ A_1\ A_2)\,[y := B] &\equiv (\oplus\ (A_1[y := B])\ (A_2[y := B])) \\
(\lambda x.A)\,[y := B] &\equiv \begin{cases} (\lambda x.A) & x \equiv y \\ (\lambda x.(A[y := B])) & x \not\equiv y\ \wedge\ x \notin \mathrm{FV}(B) \\ (\lambda z.((A[x := z])[y := B])) & x \not\equiv y \wedge \\ & z \notin \mathrm{FV}(A) \cup \mathrm{FV}(B) \end{cases}
\end{aligned}$$

In this last equation, we assume a well ordered denumerable set V of variables and choose z to be the least element such that $z \notin \mathrm{FV}(A) \cup \mathrm{FV}(B)$. From this it is easy to derive that

$$\mathrm{FV}(A[x := B]) = \begin{cases} (\mathrm{FV}(A) \setminus \{x\}) \cup \mathrm{FV}(B) & x \in \mathrm{FV}(A) \\ \mathrm{FV}(A) & x \notin \mathrm{FV}(A) \end{cases}$$

Now suppose A is a term with $y \notin \mathrm{FV}(A)$. If $\lambda x.A$ is a subterm of B, the act of replacing an occurrence of $\lambda x.A$ in B with $\lambda y.(A[x := y])$ is called *α-conversion*. If term C can be obtained from B from a finite (possibly zero) number of α-conversions, we write $C \equiv_\alpha B$ and say that C is *α-congruent* to B. For example,

$$\lambda x.(\varepsilon\ x) \equiv_\alpha \lambda y.(\varepsilon\ y)$$

From now on, we will not distinguish between α-congruent terms.

Definition 5. *Any term of the form*

$$((\lambda x.A)\ B)$$

is a β-redux *and the corresponding term*

$$A[x := B]$$

is its β-contractum. β-contracting *is the act of replacing a* β*-redux with its* β*-contractum. Given a term* D *with a* β*-redux as a subterm, if the result of replacing the subterm with its* β*-contractum results in term* E*, we say that*

$$D \rightarrow_\beta E$$

Any term of the form

$$((\oplus\ A\ B)\ C)$$

is a parallel redux *and the corresponding term*

$$(\oplus\ ((C\ True)\ A)\ ((C\ False)\ B))$$

is its parallel contractum. Parallel contracting *is the act of replacing a parallel redux with its parallel contractum. Given a term* D *with a parallel redux as a subterm, if the result of replacing the subterm with its parallel contractum results in term* E*, we say that*

$$D \rightarrow_\oplus E$$

Finally we let

$$\twoheadrightarrow \ = \ (\rightarrow_\beta \ \cup \ \rightarrow_\oplus)^*$$

If $A \twoheadrightarrow B$*, we say that* A *reduces to* B*. We define* $=$ *inductively by*

$$\begin{aligned} A \twoheadrightarrow B &\Rightarrow A = B \\ B = A &\Rightarrow A = B \\ A = C, C = B &\Rightarrow A = B \end{aligned}$$

The following theorem shows the Church-Rosser property [3] of this calculus. The complete proof can be found in the appendix.

Theorem 6. *Given* $A, B_1, B_2 \in \mathcal{T}$ *with* $A \twoheadrightarrow B_1$ *and* $A \twoheadrightarrow B_2$*, there exists a* $B_3 \in \mathcal{T}$ *with* $B_1 \twoheadrightarrow B_3$ *and* $B_2 \twoheadrightarrow B_3$*.*

From this theorem, we can derive many useful properties of our calculus. For instance, it is easy to show that if $A = B$, then there exists a C such that $A \twoheadrightarrow C$ and $B \twoheadrightarrow C$.

Given a term in our calculus, we would like to know whether or not it reduces to a configuration. The answer to this question is undecidable, as our next theorem shows. We expect such undecidability from any sufficiently expressive formalism. However, in many practical situations, we can prove that a term does reduce to a configuration. For instance, we can show that for all n, $((C_n\ (Series\ IOComp))\ IOComp)$ satisfies this property.

Theorem 7. *Let* $\mathcal{C} \subset \mathcal{T}$ *be the set of configurations. Let*

$$\mathcal{U} = \{A \in \mathcal{T} | \exists B \in \mathcal{C}. A \twoheadrightarrow B\}$$

It is undecidable whether a particular $A \in \mathcal{T}$ *belongs to* $\mathcal{U}$*.*

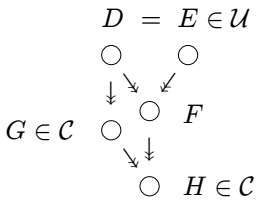
Proof. First note that there exists an injective mapping $\mathrm{gd} : \mathcal{T} \to \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers. This is called the Gödel numbering of $\mathcal{T}$. We refer the interested reader to [8], to see how one might create such a function. To show undecidability we must show that there does not exist a recursive total function $\phi : \mathbb{N} \to \mathbb{N}$ such that for all $A \in \mathcal{T}$,

$$\phi(\mathrm{gd}(A)) = \begin{cases} 0 & A \in \mathcal{U} \\ 1 & A \notin \mathcal{U} \end{cases}$$

If ϕ exists, then we call $\mathcal{U}$ *recursive*.

Now suppose $A \in \mathcal{C}$ and $A \twoheadrightarrow B$, with B an arbitrary member of $\mathcal{T}$. Since we disallow function application in configurations, $B \equiv A \in \mathcal{C}$. Thus $\mathcal{C}$ is closed under reductions.

We now show that $\mathcal{U}$ is closed under equality. Suppose $D \in \mathcal{U}$ and $D = E$, with E an arbitrary member of $\mathcal{T}$. There must exist a $F \in \mathcal{T}$ such that $D \twoheadrightarrow F$ and $E \twoheadrightarrow F$. Since $D \in \mathcal{U}$, there exists a $G \in \mathcal{C}$ with $D \twoheadrightarrow G$. By the Church-Rosser property in Theorem 7, there exists an $H \in \mathcal{T}$ with $G \twoheadrightarrow H$ and $F \twoheadrightarrow H$. Since $\mathcal{C}$ is closed under reductions, $H \in \mathcal{C}$. Thus $E \twoheadrightarrow F \twoheadrightarrow H \in \mathcal{C}$, so $E \in \mathcal{U}$. Thus $\mathcal{U}$ is closed under equality.



The Scott-Curry theorem [15] shows that in λ calculus any set of λ terms closed under equality is not recursive. It is easy to extend this theorem to our calculus, which has a superset of terms and a superset of reduction rules. We thus conclude that $\mathcal{U}$ is not recursive, so it is undecidable whether a particular $A \in \mathcal{T}$ belongs to $\mathcal{U}$. □

4 An Example Proof

To show how we might use this calculus, we give a simple proof of a claim that we made in Section 2, namely, that

$$((C_1\ (\mathit{Series}\ \mathit{IOComp}))\ \mathit{IOComp}) = \mathit{Seq}$$

To see this note that

$$
\begin{array}{ll}
 & ((C_1\ (Series\ IOComp))\ IOComp) \\
\equiv & ((Succ\ C_0)\ (Series\ IOComp))\ IOComp) \\
\equiv & (((\lambda c.\lambda f.\lambda x.(f\ ((c\ f)\ x)))\ Zero)\ (Series\ IOComp))\ IOComp) \\
\twoheadrightarrow & (((\lambda f.\lambda x.(f\ (((\lambda f.\lambda x.x)\ f)\ x)))\ (Series\ IOComp))\ IOComp) \\
\twoheadrightarrow & (((\lambda f.\lambda x.(f\ x))\ (Series\ IOComp))\ IOComp) \\
\twoheadrightarrow & ((Series\ IOComp)\ IOComp) \\
\equiv & ((\lambda a.\lambda b.\lambda in.\lambda out.(\varepsilon\ \lambda r.(\oplus\ ((a\ in)\ r)\ ((b\ r)\ out)))\ IOComp)\ IOComp) \\
\twoheadrightarrow & \lambda in.\lambda out.(\varepsilon\ \lambda r.(\oplus\ ((IOComp\ in)\ r)\ ((IOComp\ r)\ out))) \\
\equiv & Seq
\end{array}
$$

To see that this indeed reduces to a configuration, note first that for any $x, y \in V$

$$(IOComp\ x\ y) \twoheadrightarrow (\varepsilon\ (\oplus\ x\ y))$$

Then

$$Seq \twoheadrightarrow \lambda in.\lambda out.(\varepsilon\ (\lambda r.(\oplus\ (\varepsilon\ (\oplus\ in\ r))\ (\varepsilon\ (\oplus\ r\ out)))))$$

This term is closed and has no function application $(A\ B)$ as a subterm, so it is a configuration.

5 Justification of this Calculus

It may seem strange that we choose higher-order functions as the first-class citizens of our calculus rather than configurations, but as we showed above it is easy to express configurations as higher-order functions. We show that this calculus satisfies all the properties we seek in a higher-order composition language, as mentioned in the introduction:

1. Describe the interface of a configuration.
2. Describe the parallel configuration of multiple components.
3. Establish links between configurations.
4. Create new configurations through hierarchy.
5. Describe higher-order parameters to a configuration.

To see that we satisfy Property 1, note that the λ terms on the left hand side of an expression define the interface of a configuration. For instance, the interface of

$$\lambda x.\lambda y.(\varepsilon\ (\oplus\ x\ y))$$

has two parameters. When we consider a configuration, as defined in Definition 4, we will interpret all such λ terms as the ports of the configuration.

Note that when we apply one term to another, such as

$$((\lambda x.x)\ IOComp)$$

we will interpret the higher-order term on the right to represent an actual parameter of the λ term on the left.

Note that for a configuration with a parallel operator in the front, we may infer an interface. For instance, in

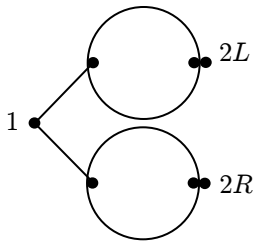
$$(\oplus\ IOComp\ IOComp)$$

we may access ports of the two $IOComp$ components by using the $Left$ and $Right$ constants of Definitions 4 and 5. This operator also endows our calculus with Property 2.

To establish links between configurations, as in Property 3, we bind their variables together. For instance

$$\lambda r.(\oplus\ (IOComp\ r)\ (IOComp\ r))$$

is used to link the first ports of the two components together. This term can be reduced to an expression with the following picture:

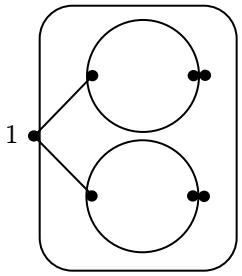


We use the L and R to note whether we need to access this port via a $Left$ or $Right$ term.

Should we wish to hide these ports, we can use

$$\lambda r.(\varepsilon\ (\oplus\ (IOComp\ r)\ (IOComp\ r)))$$

which will reduce to an expression with this picture:



In this way, our encapsulation operator gives us hierarchy (Property 4).

Finally, by using λ calculus, we are able to satisfy Property 5. We simply view a higher-order function in the calculus as a configuration with higher-order parameters. While it may not reduce to a configuration for all parameters, this is not really a problem; it simply reflects the fact that the parameterized configuration may not be well defined for all actual parameters.

6 Conclusions

We have introduced a calculus for higher-order composition languages. Through examples, we have demonstrated the usefulness of this extended λ calculus in constructing configurations with higher-order parameters. We have detailed the formalism and a few of its implications, and showed that this calculus can serve as a formalism for higher-order composition languages. With this calculus, the Ptalon project has a mathematical framework to guide future work in constructing a concrete higher-order composition language.

7 Acknowledgements

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF award #CCR-0225610), the State of California Micro Program, and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft, and Toyota. In addition, we would like to thank Jörn Janneck, Jie Liu, Stephen Neuendorffer, and Ed Willink; they have all been part of the larger discussion on higher-order composition languages.

A A Proof of the Church-Rosser Property

This paper introduced a calculus for higher-order composition languages. This calculus is an extension of the standard λ calculus [3]. This appendix contains a proof of the Church-Rosser property for this extension.

Definition 1. *Let V be a denumerable (countable and infinite) set of* variables. *We build a set $\mathcal{T}$ of* model terms *inductively:*

$$\begin{aligned} x \in V &\Rightarrow x \in \mathcal{T} && (1)\\ A \in \mathcal{T} &\Rightarrow (\varepsilon\ A) \in \mathcal{T} && (2)\\ A, B \in \mathcal{T} &\Rightarrow (A\ B) \in \mathcal{T} && (3)\\ &\quad (\oplus\ A\ B) \in \mathcal{T} && (4)\\ A \in \mathcal{T}, x \in V &\Rightarrow \lambda x.A \in \mathcal{T} && (5) \end{aligned}$$

We will often omit parentheses. We note that application (Rule 3) is left associative, whereas abstraction (Rule 5) is right associative.

Definition 2. *We define $A[x := B]$ inductively by*

$$\begin{aligned} x[y := B] &\equiv \begin{cases} B & x \equiv y \\ x & x \not\equiv y \end{cases}\\ (\varepsilon\ A)[y := B] &\equiv (\varepsilon\ A[y := B])\\ (A_1\ A_2)[y := B] &\equiv ((A_1[y := B])\ (A_2[y := B]))\\ (\oplus\ A_1\ A_2)[y := B] &\equiv (\oplus\ (A_1[y := B])\ (A_2[y := B]))\\ (\lambda x.A)[y := B] &\equiv \begin{cases} (\lambda x.A) & x \equiv y \\ (\lambda x.(A[y := B])) & x \not\equiv y \wedge x \notin \mathrm{FV}(B) \\ (\lambda z.((A[x := z])[y := B])) & x \not\equiv y \wedge \\ & z \notin \mathrm{FV}(A) \cup \mathrm{FV}(B) \end{cases} \end{aligned}$$

Definition 3. *Any term of the form*

$$((\lambda x.A)\ B)$$

is a β-redux *and the corresponding term*

$$A[x := B]$$

is its β-contractum. β-contracting *is the act of replacing a β-redux with its β-contractum. Given a term D with a β-redux as a subterm, if the result of replacing the subterm with its β-contractum results in term E, we say that*

$$D \rightarrow_\beta E$$

Any term of the form

$$((\oplus\ A\ B)\ C)$$

is a parallel redux *and the corresponding term*

$$(\oplus\ ((C\ (\lambda x.\lambda y.x))\ A)\ ((C\ (\lambda x.\lambda y.y))\ B))$$

is its parallel contractum. Parallel contracting *is the act of replacing a parallel redux with its parallel contractum. To simplify the parallel contractum, let*

$$\begin{aligned} True &\equiv \lambda x.\lambda y.x \\ False &\equiv \lambda x.\lambda y.y \end{aligned}$$

Then the parallel contractum becomes

$$(\oplus\ ((C\ True)\ A)\ ((C\ False)\ B))$$

Given a term D with a parallel redux as a subterm, if the result of replacing the subterm with its parallel contractum results in term E, we say that

$$D \rightarrow_{\oplus} E$$

Finally we let

$$\twoheadrightarrow\ =\ (\rightarrow_{\beta}\ \cup\ \rightarrow_{\oplus})^{*}$$

If $A \twoheadrightarrow B$, we say that A reduces to B. We define $=$ inductively by

$$\begin{aligned} A \twoheadrightarrow B &\Rightarrow A = B \\ B = A &\Rightarrow A = B \\ A = C, C = B &\Rightarrow A = B \end{aligned}$$

The next definition is modeled after [3]. The purpose is to eventually extend the Church-Rosser theorem to our calculus.

Definition 4. *The set $\underline{\mathcal{T}}$ is defined inductively by:*

$$\begin{aligned} x \in V &\Rightarrow x \in \underline{\mathcal{T}} \\ A \in \underline{\mathcal{T}} &\Rightarrow (\varepsilon\ A) \in \underline{\mathcal{T}} \\ A, B \in \underline{\mathcal{T}} &\Rightarrow (A\ B) \in \underline{\mathcal{T}} \\ &\quad\ (\oplus\ A\ B) \in \underline{\mathcal{T}} \\ A, B, C \in \underline{\mathcal{T}} &\Rightarrow ((\underline{\oplus}\ A\ B)\ C) \in \underline{\mathcal{T}} \\ A \in \underline{\mathcal{T}}, x \in V &\Rightarrow \lambda x.A \in \underline{\mathcal{T}} \\ A, B \in \underline{\mathcal{T}}, x \in V &\Rightarrow ((\underline{\lambda} x.A)\ B) \in \underline{\mathcal{T}} \end{aligned}$$

For terms in $\underline{\mathcal{T}}$ we can extend the definition of substitution, as in $A[x := B]$, in the obvious way. In particular, we treat underlined versions of $\underline{\lambda}$ and $\underline{\oplus}$ no different than λ and $\oplus$ in Definition 2.

Definition 5. *In $\underline{\mathcal{T}}$, any term of the form*

$$((\lambda x.A)\ B)$$

or

$$((\underline{\lambda}x.A)\ B)$$

is a β-redux *and the corresponding term*

$$A[x := B]$$

is its β-contractum. β-contracting *is the act of replacing a* β*-redux with its* β*-contractum. Given a term* D *with a* β*-redux as a subterm, if the result of replacing the subterm with its* β*-contractum results in term* E*, we say that*

$$D \rightarrow_{\underline{\beta}} E$$

Any term of the form

$$((\oplus\ A\ B)\ C)$$

or

$$((\underline{\oplus}\ A\ B)\ C)$$

is a parallel redux *and the corresponding term*

$$(\oplus\ ((C\ True)\ A)\ ((C\ False)\ B))$$

is its parallel contractum. Parallel contracting *is the act of replacing a parallel redux with its parallel contractum. Given a term* D *with a parallel redux as a subterm, if the result of replacing the subterm with its parallel contractum results in term* E*, we say that*

$$D \rightarrow_{\underline{\oplus}} E$$

Finally we let

$$\twoheadrightarrow_{\underline{\mathcal{T}}} \ = \ (\rightarrow_{\underline{\beta}} \cup \rightarrow_{\underline{\oplus}})^*$$

If $A \twoheadrightarrow_{\underline{\mathcal{T}}} B$*, we say that* A *reduces to* B*.*

Definition 6. *Let* $|\cdot| : \underline{\mathcal{T}} \rightarrow \mathcal{T}$ *be defined so that* $|A|$ *is equivalent to* A *with each occurrence of* $\underline{\lambda}$ *replaced with* λ *and each occurrence of* $\underline{\oplus}$ *replaced with* $\oplus$*.*

Definition 7. *We define* $\phi : \underline{\mathcal{T}} \rightarrow \mathcal{T}$ *inductively with*

$$\begin{array}{rcl}
\phi(x) & \equiv & x \\
\phi((\varepsilon\ A)) & \equiv & (\varepsilon\ \phi(A)) \\
\phi((A\ B)) & \equiv & (\phi(A)\ \phi(B)) \\
\phi((\oplus\ A\ B)) & \equiv & (\oplus\ \phi(A)\ \phi(B)) \\
\phi(((\underline{\oplus}\ A\ B)\ C)) & \equiv & (\oplus\ ((\phi(C)\ True)\ \phi(A))\ ((\phi(C)\ False)\ \phi(B))) \\
\phi(\lambda x.A) & \equiv & (\lambda x.\phi(A)) \\
\phi(((\underline{\lambda}x.A)\ B)) & \equiv & \phi(A)[x := \phi(B)]
\end{array}$$

Lemma 8. *Given* $A' \in \underline{\mathcal{T}}$ *and* $B \in \mathcal{T}$*, if* $|A'| \twoheadrightarrow B$*, then there exists a* $B' \in \underline{\mathcal{T}}$ *such that* $A' \twoheadrightarrow_{\underline{\mathcal{T}}} B'$ *and* $|B'| \equiv B$*.*

Proof. First suppose $|A'| \to_\beta B$ or $|A'| \to_\oplus B$. Then B' can be obtained from A' by contracting the corresponding redux. The result follows from transitivity. $\square$

Lemma 9. *Given $x, y \in V$ with $x \not\equiv y$ and $A, B, C \in \mathcal{T}$ with $x \notin FV(C)$ then*

$$(A[x := B])[y := C] \equiv (A[y := C])[x := B[y := C]]$$

Proof. We prove this with induction on the structure of A.

1. Case: $A \equiv x$. Then

$$\begin{aligned}(A[x := B])[y := C] &\equiv B[y := C] \\ &\equiv A[x := B[y := C]] \\ &\equiv (A[y := C])[x := B[y := C]]\end{aligned}$$

2. Case: $A \equiv y$. Then

$$\begin{aligned}(A[x := B])[y := C] &\equiv A[y := C] \\ &\equiv C \\ &\equiv C[x := B[y := C]] \\ &\equiv (A[y := C])[x := B[y := C]]\end{aligned}$$

3. Case: $A \equiv z$ where $z \not\equiv x$ and $z \not\equiv y$. Then

$$\begin{aligned}(A[x := B])[y := C] &\equiv A \\ &\equiv A[x := B[y := C]] \\ &\equiv (A[y := C])[x := B[y := C]]\end{aligned}$$

4. Case: $A \equiv (\varepsilon\ A_1)$, for some A_1. By the inductive hypothesis, the result holds for A_1. Then

$$\begin{aligned}(A[x := B])[y := C] &\equiv ((\varepsilon\ A_1)[x := B])[y := C] \\ &\equiv (\varepsilon\ (A_1[x := B])[y := C]) \\ &\equiv (\varepsilon\ (A_1[y := C])[x := B[y := C]]) \\ &\equiv ((\varepsilon\ A_1)[y := C])[x := B[y := C]] \\ &\equiv (A[y := C])[x := B[y := C]]\end{aligned}$$

5. Case: $A \equiv (A_1\ A_2)$, for some A_1 and A_2. By the inductive hypothesis, the result holds for A_1 and A_2. Then

$$\begin{aligned}(A[x := B])[y := C] &\equiv ((A_1\ A_2)[x := B])[y := C] \\ &\equiv ((A_1[x := B])[y := C]\ (A_2[x := B])[y := C]) \\ &\equiv ((A_1[y := C])[x := B[y := C]] \\ &\qquad (A_2[y := C])[x := B[y := C]]) \\ &\equiv ((A_1\ A_2)[y := C])[x := B[y := C]] \\ &\equiv (A[y := C])[x := B[y := C]]\end{aligned}$$

6. Case: $A \equiv (\oplus\ A_1\ A_2)$, for some A_1 and A_2. By the inductive hypothesis, the result holds for A_1 and A_2. Then

$$\begin{array}{rcl}
(A[x := B])[y := C] & \equiv & ((\oplus\ A_1\ A_2)[x := B])[y := C] \\
& \equiv & (\oplus\ (A_1[x := B])[y := C]\ (A_2[x := B])[y := C]) \\
& \equiv & (\oplus\ (A_1[y := C])[x := B[y := C]] \\
& & \quad (A_2[y := C])[x := B[y := C]]) \\
& \equiv & ((\oplus\ A_1\ A_2)[y := C])[x := B[y := C]] \\
& \equiv & (A[y := C])[x := B[y := C]]
\end{array}$$

7. Case: $A \equiv \lambda z.A_1$ where $z \in V$ and $z \equiv x$. By the inductive hypothesis, the result holds for A_1.

$$\begin{array}{rcl}
(A[x := B])[y := C] & \equiv & ((\lambda z.A_1)[x := B])[y := C] \\
& \equiv & ((\lambda x.A_1)[x := B])[y := C] \\
& \equiv & (\lambda x.A_1)[y := C] \\
& \equiv & \lambda x.A_1[y := C] \\
& \equiv & (\lambda x.A_1[y := C])[x := B[y := C]] \\
& \equiv & ((\lambda x.A_1)[y := C])[x := B[y := C]] \\
& \equiv & (A[y := C])[x := B[y := C]]
\end{array}$$

8. Case: $A \equiv \lambda z.A_1$ where $z \in V$ and $z \equiv y$ and $y \notin \mathrm{FV}(B)$. By the inductive hypothesis, the result holds for A_1.

$$\begin{array}{rcl}
(A[x := B])[y := C] & \equiv & ((\lambda z.A_1)[x := B])[y := C] \\
& \equiv & ((\lambda y.A_1)[x := B])[y := C] \\
& \equiv & (\lambda y.A_1[x := B]) \\
& \equiv & (\lambda y.A_1[x := B[y := C]]) \\
& \equiv & (\lambda y.A_1)[x := B[y := C]] \\
& \equiv & ((\lambda y.A_1)[y := C])[x := B[y := C]] \\
& \equiv & (A[y := C])[x := B[y := C]]
\end{array}$$

9. Case: $A \equiv \lambda z.A_1$ where $z \in V$ and $z \equiv y$ and $y \in \mathrm{FV}(B)$. By the inductive hypothesis, the result holds for A_1. Let w be some variable such

that $w \notin \mathrm{FV}(A_1) \cup \mathrm{FV}(B)$. Then

$$\begin{aligned}
(A[x := B])[y := C] &\equiv ((\lambda z.A_1)[x := B])[y := C] \\
&\equiv ((\lambda y.A_1)[x := B])[y := C] \\
&\equiv (\lambda y.A_1[x := B])[y := C] \\
&\equiv (\lambda w.A_1[x := B])[y := C] \\
&\equiv (\lambda w.(A_1[x := B])[y := C]) \\
&\equiv ((\lambda w.A_1[y := C])[x := B[y := C]]) \\
&\equiv ((\lambda w.A_1)[y := C])[x := B[y := C]] \\
&\equiv (A[y := C])[x := B[y := C]]
\end{aligned}$$

10. Case: $A \equiv \lambda z.A_1$ where $z \in V$ and $z \not\equiv y$ and $z \not\equiv x$. By the inductive hypothesis, the result holds for A_1.

$$\begin{aligned}
(A[x := B])[y := C] &\equiv ((\lambda z.A_1)[x := B])[y := C] \\
&\equiv ((\lambda z.A_1[x := B])[y := C]) \\
&\equiv ((\lambda z.A_1[y := C])[x := B[y := C]]) \\
&\equiv ((\lambda z.A_1)[y := C])[x := B[y := C]] \\
&\equiv (A[y := C])[x := B[y := C]]
\end{aligned}$$

□

Lemma 10. *For all $A, B \in \underline{\mathcal{T}}$ and $x \in V$,*

$$\phi(A[x := B]) = \phi(A)[x := \phi(B)]$$

Proof. We prove this with induction on the structure of A.

1. Case: $A \equiv x$. Then

$$\begin{aligned}
\phi(A[x := B]) &\equiv \phi(B) \\
&\equiv x[x := \phi(B)] \\
&\equiv \phi(x)[x := \phi(B)] \\
&\equiv \phi(A)[x := \phi(B)]
\end{aligned}$$

2. Case: $A \equiv y \not\equiv x$. Then

$$\begin{aligned}
\phi(A[x := B]) &\equiv \phi(y) \\
&\equiv y \\
&\equiv y[x := \phi(B)] \\
&\equiv \phi(y)[x := \phi(B)] \\
&\equiv \phi(A)[x := \phi(B)]
\end{aligned}$$

3. Case: $A \equiv (\varepsilon\ C)$, for some C. By the inductive hypothesis, the result holds for C. Then

$$\begin{aligned}
\phi(A[x := B]) &\equiv \phi((\varepsilon\ C[x := B])) \\
&\equiv (\varepsilon\ \phi(C[x := B])) \\
&\equiv (\varepsilon\ \phi(C)[x := \phi(B)]) \\
&\equiv (\varepsilon\ \phi(C))[x := \phi(B)] \\
&\equiv \phi((\varepsilon\ C))[x := \phi(B)] \\
&\equiv \phi(A)[x := \phi(B)]
\end{aligned}$$

4. Case: $A \equiv (C\ D)$, for some C and D. By the inductive hypothesis, the result holds for C and D. Then

$$\begin{aligned}
\phi(A[x := B]) &\equiv \phi((C\ D)[x := B]) \\
&\equiv \phi((C[x := B]\ D[x := B])) \\
&\equiv (\phi(C[x := B])\ \phi(D[x := B])) \\
&\equiv (\phi(C)[x := \phi(B)]\ \phi(D)[x := \phi(B)]) \\
&\equiv (\phi(C)\ \phi(D))[x := \phi(B)] \\
&\equiv \phi((C\ D))[x := \phi(B)] \\
&\equiv \phi(A)[x := \phi(B)]
\end{aligned}$$

5. Case: $A \equiv (\oplus\ C\ D)$, for some C and D. By the inductive hypothesis, the result holds for C and D. Then

$$\begin{aligned}
\phi(A[x := B]) &\equiv \phi((\oplus\ C\ D)[x := B]) \\
&\equiv \phi((\oplus\ C[x := B]\ D[x := B])) \\
&\equiv (\oplus\ \phi(C[x := B])\ \phi(D[x := B])) \\
&\equiv (\oplus\ \phi(C)[x := \phi(B)]\ \phi(D)[x := \phi(B)]) \\
&\equiv (\oplus\ \phi(C)\ \phi(D))[x := \phi(B)] \\
&\equiv \phi((\oplus\ C\ D))[x := \phi(B)] \\
&\equiv \phi(A)[x := \phi(B)]
\end{aligned}$$

6. Case: $A \equiv ((\underline{\oplus}\ C\ D)\ E)$ for some C, D, and E. By the inductive hypoth-

esis, the result holds for C, D, and E. Then

$$\begin{aligned}
\phi(A[x := B]) &\equiv \phi(((\underline{\oplus}\ C\ D)\ E)[x := B]) \\
&\equiv \phi(((\underline{\oplus}\ C\ D)[x := B]\ E[x := B])) \\
&\equiv \phi(((\underline{\oplus}\ C[x := B]\ D[x := B])\ E[x := B])) \\
&\equiv (\oplus\ ((\phi(E[x := B])\ True)\ \phi(C[x := B])) \\
&\qquad ((\phi(E[x := B])\ False)\ \phi(D[x := B]))) \\
&\equiv (\oplus\ ((\phi(E)[x := \phi(B)]\ True)\ \phi(C)[x := \phi(B)]) \\
&\qquad ((\phi(E)[x := \phi(B)]\ False)\ \phi(D)[x := \phi(B)])) \\
&\equiv (\oplus\ ((\phi(E)\ True)[x := \phi(B)]\ \phi(C)[x := \phi(B)]) \\
&\qquad ((\phi(E)\ False)[x := \phi(B)]\ \phi(D)[x := \phi(B)])) \\
&\equiv (\oplus\ ((\phi(E)\ True)\ \phi(C))[x := \phi(B)] \\
&\qquad ((\phi(E)\ False)\ \phi(D))[x := \phi(B)]) \\
&\equiv (\oplus\ (((\phi(E)\ True)\ \phi(C)) \\
&\qquad (((\phi(E)\ False)\ \phi(D)))[x := \phi(B)] \\
&\equiv \phi(((\underline{\oplus}\ C\ D)\ E))[x := \phi(B)] \\
&\equiv \phi(A)[x := \phi(B)]
\end{aligned}$$

7. Case: $A \equiv \lambda x.C$. By the inductive hypothesis, the result holds for C. Let z be some variable not in $\mathrm{FV}(C)$ or $\mathrm{FV}(B)$. Then

$$\begin{aligned}
\phi(A[x := B]) &\equiv \phi((\lambda x.C)[x := B]) \\
&\equiv \phi(\lambda x.C) \\
&\equiv (\lambda x.\phi(C)) \\
&\equiv (\lambda x.\phi(C))[x := \phi(B)] \\
&\equiv \phi(A)[x := \phi(B)]
\end{aligned}$$

8. Case: $A \equiv \lambda y.C$ where $y \not\equiv x$. By the inductive hypothesis, the result holds for C. Suppose first that $y \in \mathrm{FV}(B)$, and let z be some variable not in $\mathrm{FV}(C)$ or $\mathrm{FV}(B)$. Since the result holds for C it holds for $C[y := z]$ in this case. Then

$$\begin{aligned}
\phi(A[x := B]) &\equiv \phi((\lambda y.C)[x := B]) \\
&\equiv \phi((\lambda z.((C[y := z])[x := B]))) \\
&\equiv (\lambda z.\phi((C[y := z])[x := B])) \\
&\equiv (\lambda z.\phi(C[y := z])[x := \phi(B)]) \\
&\equiv (\lambda z.\phi(C)[y := z])[x := \phi(B)] \\
&\equiv (\lambda y.\phi(C))[x := \phi(B)] \\
&\equiv \phi(\lambda y.C)[x := \phi(B)] \\
&\equiv \phi(A)[x := \phi(B)]
\end{aligned}$$

Now if $y \notin \mathrm{FV}(B)$, we simply have

$$\begin{aligned}
\phi(A[x := B]) &\equiv \phi((\lambda y.C)[x := B]) \\
&\equiv \phi((\lambda y.C[x := B])) \\
&\equiv (\lambda y.\phi(C[x := B])) \\
&\equiv (\lambda y.\phi(C)[x := \phi(B)]) \\
&\equiv (\lambda y.\phi(C))[x := \phi(B)] \\
&\equiv \phi(\lambda y.C)[x := \phi(B)] \\
&\equiv \phi(A)[x := \phi(B)]
\end{aligned}$$

9. Case: $A \equiv ((\underline{\lambda} x.C)\ D)$. By the inductive hypothesis, the result holds for C and D. Then

$$\begin{aligned}
\phi(A[x := B]) &\equiv \phi(((\underline{\lambda} x.C)\ D)[x := B]) \\
&\equiv \phi(((\underline{\lambda} x.C)[x := B]\ D[x := B])) \\
&\equiv \phi(((\underline{\lambda} x.C)\ D[x := B])) \\
&\equiv \phi(C)[x := \phi(D[x := B])] \\
&\equiv \phi(C)[x := \phi(D)[x := \phi(B)]] \\
&\equiv (\phi(C)[x := \phi(D)])[x := \phi(B)] \\
&\equiv \phi(((\underline{\lambda} x.C)\ D))[x := \phi(B)] \\
&\equiv \phi(A)[x := \phi(B)]
\end{aligned}$$

10. Case: $A \equiv ((\underline{\lambda} y.C)\ D)$, where $y \not\equiv x$. By the inductive hypothesis, the result holds for C and D. Suppose first that $y \in \mathrm{FV}(B)$ and $z \notin \mathrm{FV}(C)$ or $\mathrm{FV}(B)$. In this case, the result will also hold for $C[y := z]$. Then

$$\begin{aligned}
\phi(A[x := B)) &\equiv \phi(((\underline{\lambda} y.C)\ D)[x := B]) \\
&\equiv \phi(((\underline{\lambda} y.C)[x := B]\ D[x := B])) \\
&\equiv \phi(((\underline{\lambda} z.((C[y := z])[x := B]))\ D[x := B])) \\
&\equiv \phi((C[y := z])[x := B])[z := \phi(D[x := B])] \\
&\equiv (\phi(C[y := z])[x := \phi(B)])[z := \phi(D)[x := \phi(B)]] \\
&\equiv (\phi(C[y := z])[z := \phi(D)])[x := \phi(B)] \\
&\equiv ((\phi(C)[y := z])[z := \phi(D)])[x := \phi(B)] \\
&\equiv (\phi(C)[y := \phi(D)])[x := \phi(B)] \\
&\equiv \phi(((\underline{\lambda} y.C)\ D))[x := \phi(B)] \\
&\equiv \phi(A)[x := \phi(B)]
\end{aligned}$$

Here we used the substitution lemma (Lemma 9).

Now if $y \notin \mathrm{FV}(B)$, we simply have

$$\begin{array}{rcl}
\phi(A[x := B]) & \equiv & \phi(((\underline{\lambda} y.C)\ D)[x := B]) \\
& \equiv & \phi(((\underline{\lambda} y.C)[x := B]\ D[x := B])) \\
& \equiv & \phi(((\underline{\lambda} y.C[x := B])\ D[x := B])) \\
& \equiv & \phi(C[x := B])[y := \phi(D[x := B])] \\
& \equiv & (\phi(C)[x := \phi(B)])[y := \phi(D)[x := \phi(B)]] \\
& \equiv & (\phi(C)[y := \phi(D)])[x := \phi(B)] \\
& \equiv & \phi(((\underline{\lambda} y.C)\ D))[x := \phi(B)] \\
& \equiv & \phi(A)[x := \phi(B)]
\end{array}$$

Again, we used the substitution lemma.

□

Lemma 11. *Given $A, B \in \underline{\mathcal{T}}$, if $A \twoheadrightarrow_{\underline{\mathcal{T}}} B$, then $\phi(A) \twoheadrightarrow \phi(B)$.*

Proof. We prove this by induction on the structure of $\twoheadrightarrow_{\underline{\mathcal{T}}}$.

1. Case: $A \to_\beta B$ occurs from contracting a redux $((\lambda x.C)\ D)$ which is a subterm of $\overline{A}$. By the previous lemma,

   $$\phi(C[x := D]) \equiv \phi(C)[x := \phi(D)]$$

   so contracting the redux $((\lambda x.\phi(C))\ \phi(D))$ in $\phi(A)$ will give $\phi(B)$.

2. Case: $A \to_\beta B$ occurs from contracting a redux $((\underline{\lambda} x.C)\ D)$ which is a subterm of $\overline{A}$. Then $\phi(A) \equiv \phi(B)$.

3. Case: $A \to_{\oplus} B$ occurs from contracting a redux $((\oplus\ C\ D)\ E)$ which is a subterm of $\overline{A}$. Then contracting the redux $((\oplus\ \phi(C)\ \phi(D))\ \phi(E))$ in $\phi(A)$ will give $\phi(B)$.

4. Case: $A \to_{\oplus} B$ occurs from contracting a redux $((\underline{\oplus}\ C\ D)\ E)$ which is a subterm of $\overline{A}$. Then $\phi(A) \equiv \phi(B)$.

The lemma then follows from transitivity. □

Lemma 12. *For all $A \in \underline{\mathcal{T}}$, $|A| \twoheadrightarrow \phi(A)$.*

Proof. We prove this by induction on the structure of A.

1. Case: $A \equiv x$. In this case, $|A| \equiv x \equiv \phi(A)$, so the result is trivial.

2. Case: $A \equiv (\varepsilon\ B)$, where the result holds for B by the inductive hypothesis. Then

   $$\begin{array}{rcl}
   |A| & \equiv & (\varepsilon\ |B|) \\
   & \twoheadrightarrow & (\varepsilon\ \phi(B)) \\
   & \equiv & \phi(A)
   \end{array}$$

3. Case: $A \equiv (B\ C)$, where the result holds for B and C by the inductive hypothesis. Then

$$\begin{array}{rcl} |A| & \equiv & (|B|\ |C|) \\ & \twoheadrightarrow & (\phi(B)\ \phi(C)) \\ & \equiv & \phi(A) \end{array}$$

4. Case: $A \equiv (\oplus\ B\ C)$, where the result holds for B and C by the inductive hypothesis. Then

$$\begin{array}{rcl} |A| & \equiv & (\oplus\ |B|\ |C|) \\ & \twoheadrightarrow & (\oplus\ \phi(B)\ \phi(C)) \\ & \equiv & \phi(A) \end{array}$$

5. Case: $A \equiv ((\underline{\oplus}\ B\ C)\ D)$, where the result holds for B, C, and D by the inductive hypothesis. Then

$$\begin{array}{rcl} |A| & \equiv & |((\underline{\oplus}\ B\ C)\ D)| \\ & \equiv & ((\oplus\ |B|\ |C|)\ |D|) \\ & \twoheadrightarrow & ((\oplus\ \phi(B)\ \phi(C))\ \phi(D)) \\ & \twoheadrightarrow & ((\oplus\ ((\phi(D)\ True)\ \phi(B))\ ((\phi(D)\ False)\ \phi(C)) \\ & \equiv & \phi(((\underline{\oplus}\ B\ C)\ D)) \\ & \equiv & \phi(A) \end{array}$$

6. Case: $A \equiv (\lambda x.B)$, where the result holds for B by the inductive hypothesis. Then

$$\begin{array}{rcl} |A| & \equiv & (\lambda x.|B|) \\ & \twoheadrightarrow & (\lambda x.\phi(B)) \\ & \equiv & \phi(A) \end{array}$$

7. Case: $A \equiv ((\underline{\lambda} x.B)\ C)$, where the result holds for B, C, and D by the inductive hypothesis. Then

$$\begin{array}{rcl} |A| & \equiv & ((\lambda x.|B|)\ |C|) \\ & \twoheadrightarrow & ((\lambda x.\phi(B))\ \phi(C)) \\ & \twoheadrightarrow & \phi(B)[x := \phi(C)] \\ & \equiv & \phi(((\underline{\lambda} x.B)\ C)) \\ & \equiv & \phi(A) \end{array}$$

□

Lemma 13. *Given $A, B_1, B_2 \in \mathcal{T}$ with $A \rightarrow_\beta B_1$ and $A \twoheadrightarrow B_2$, there exists a $B_3 \in \mathcal{T}$ with $B_1 \twoheadrightarrow B_3$ and $B_2 \twoheadrightarrow B_3$.*

Proof. Let B_1 be the result of contracting the redux occurrence $R \equiv ((\lambda x.C)\ D)$ in A. Create $A' \in \underline{\mathcal{T}}$ by replacing R in A with $R' \equiv ((\underline{\lambda} x.C)\ D)$. Then $|A'| \equiv A$ and $\phi(A') \equiv B_1$. By Lemma 8, there exists $B_2' \in \underline{\mathcal{T}}$ such that $A' \twoheadrightarrow_{\underline{\mathcal{T}}} B_2'$ and $|B_2'| \equiv B_2$.

Let $B_3 \equiv \phi(B_2')$. Then $B_2 \twoheadrightarrow B_3$ by Lemma 12, and $B_1 \twoheadrightarrow B_3$ by Lemma 11. □

Lemma 14. *Given $A, B_1, B_2 \in \mathcal{T}$ with $A \rightarrow_{\oplus} B_1$ and $A \twoheadrightarrow B_2$, there exists a $B_3 \in \mathcal{T}$ with $B_1 \twoheadrightarrow B_3$ and $B_2 \twoheadrightarrow B_3$.*

Proof. Let B_1 be the result of contracting the redux occurrence $R \equiv ((\oplus\ C\ D)\ E)$ in A. Create $A' \in \underline{\mathcal{T}}$ by replacing R in A with $R' \equiv ((\underline{\oplus}\ C\ D)\ E)$. Then $|A'| \equiv A$ and $\phi(A') \equiv B_1$. By Lemma 8, there exists $B_2' \in \underline{\mathcal{T}}$ such that $A' \twoheadrightarrow_{\underline{\mathcal{T}}} B_2'$ and $|B_2'| \equiv B_2$.

Let $B_3 \equiv \phi(B_2')$. Then $B_2 \twoheadrightarrow B_3$ by Lemma 12, and $B_1 \twoheadrightarrow B_3$ by Lemma 11. □

Theorem 15. *Given $A, B_1, B_2 \in \mathcal{T}$ with $A \twoheadrightarrow B_1$ and $A \twoheadrightarrow B_2$, there exists a $B_3 \in \mathcal{T}$ with $B_1 \twoheadrightarrow B_3$ and $B_2 \twoheadrightarrow B_3$.*

Proof. If $A \twoheadrightarrow B_1$, there exists a sequence $(A_0, \ldots, A_n)$ in $\mathcal{T}$ such that for each $i < n$, either $A_0 \rightarrow_{\beta} A_{i+1}$ or $A_i \rightarrow_{\oplus} A_{i+1}$, and $A \equiv A_0$ and $B_1 \equiv A_n$. We now perform an induction over the length of this sequence.

1. Case: $B_1 \equiv A_0 \equiv A$. In this case the result is trivial. We choose $B_3 \equiv B_2 \equiv B_1 \equiv A$.

2. Case: $B_1 \equiv A_{i+1}$, where by the inductive hypothesis, there is a B_3' such that $A_i \twoheadrightarrow B_3'$ and $B_2 \twoheadrightarrow B_3'$. If $A_i \rightarrow_{\beta} A_{i+1}$, then from Lemma 13, there exists a B_3 such that $B_3' \twoheadrightarrow B_3$ and $B_1 \twoheadrightarrow B_3$. If $A_i \rightarrow_{\oplus} A_{i+1}$, then from Lemma 14, there exists a B_3 such that $B_3' \twoheadrightarrow B_3$ and $B_1 \twoheadrightarrow B_3$. In either case, $B_2 \twoheadrightarrow B_3$ by transitivity.

□

References

[1] J. Aynsley and D. Long. Draft standard SystemC language reference manual. Technical report, Open SystemC Initiative, 2005.

[2] H. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. Number 103 in Studies in Logic and the Foundations of Mathematics. North-Holland, 1981. Second edition, 1984.

[3] H. Barendregt and E. Barendsen. Introduction to lambda calculus (1994). Technical report, Department of Computer Science, Catholic University of Nijmegen, Toernooiveld, 1, 6525 ED Nijmegen, The Nederlands, July 1991.

[4] A. Benveniste and G. Berry. *The synchronous approach to reactive and real-time systems*, pages 147–159. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[5] A. Benveniste, B. Caillaud, L. P. Carloni, and A. Sangiovanni-Vincentelli. Tag machines. In *EMSOFT '05: Proceedings of the 5th ACM International Conference on Embedded Software*, pages 255–263, New York, NY, USA, 2005. ACM Press.

[6] Bluespec. Automatic generation of control logic with Bluespec SystemVerilog. Technical report, Bluespec Inc., February 2005.

[7] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in Java (Volume 1: Introduction to Ptolemy II). Technical Report UCB/ERL M05/21, EECS, University of California, Berkeley, 2005.

[8] A. Church. An unsolvable problem in elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[9] J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a higher-order synchronous data-flow language. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 230–239, New York, NY, USA, 2004. ACM Press.

[10] J. B. Dabney and T. L. Harman. *Mastering SIMULINK*. Prentice Hall Professional Technical Reference, 2003.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI)*, San Francisco, CA, 2004.

[12] S. A. Edwards and O. Tardieu. SHIM: a deterministic model for heterogeneous embedded systems. In *EMSOFT '05: Proceedings of the 5th ACM International Conference on Embedded Software*, pages 264–272, New York, NY, USA, 2005. ACM Press.

[13] J. Eker and J. Janneck. CAL language report: Specification of the CAL actor language. Technical Report UCB/ERL M03/48, EECS, University of California, Berkeley, 2003.

[14] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, 2003.

[15] R. Hindley and J. Seldin. *Introduction to Combinators and λ-Calculus*. Students Texts Nr. 1. London Mathematical Society, 1986.

[16] J. W. Janneck and R. Esser. Higher-order petri net modeling—techniques and applications. In *Workshop on Software Engineering and Formal Methods*, 2002.

[17] G. W. Johnson and R. Jennings. *LabVIEW Graphical Programming*. McGraw-Hill Professional, 2001.

[18] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Y. Thomason IV, G. G. Nordstrom, and P. Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing*, Budapest, Hungary, May 2001.

[19] E. A. Lee. Model-driven development - from object-oriented design to actor-oriented design. In *Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation (a.k.a. The Monterey Workshop)*, Chicago, 2003.

[20] E. A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, January 10 2006.

[21] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on CAD*, 17(12), 1998.

[22] J. McCluskey. Practical applications of recursive VHDL components in FPGA synthesis. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, page 252, New York, NY, USA, 1999. ACM Press.

[23] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[24] R. Milner. Flowgraphs and flow algebras. *J. ACM*, 26(4):794–818, 1979.

[25] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[26] G. G. Nordstrom. *Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments*. PhD thesis, Vanderbilt University, 1999.

[27] G. A. Papadopoulos and F. Arbab. Coordination models and languages. In M. V. Zelkowitz, editor, *Advances in Computers*, volume 46, pages 329–400. Academic Press, 1998.

[28] D. L. Perry. *VHDL (2nd ed.)*. McGraw-Hill, Inc., New York, NY, USA, 1993.

[29] H. J. Reekie. *Realtime Signal Processing - Dataflow, Visual, and Functional Programming*. PhD thesis, University of Technology at Sydney, 1995.

[30] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. The liberty simulation environment, version 1.0. *SIGMETRICS Perform. Eval. Rev.*, 31(4):19–24, 2004.

[31] H. L. Vangheluwe, J. de Lara, and P. J. Mosterman. An introduction to multi-paradigm modelling and simulation. In F. Barros and N. Giambiasi, editors, *Proceedings of the AIS 2002 Conference (AI, Simulation and Planning in High Autonomy Systems)*, pages 9–20, Lisboa, Portugal, 2002.