The Power of Higher-Order Composition Languages in System Design

James Adam Cataldo

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-189
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-189.html

December 18, 2006

The Power of Higher-Order Composition Languages in System Design

by

James Adam Cataldo

B.S. (Washington University in St. Louis) 2001
M.S. (University of California, Berkeley) 2003

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Electrical Engineering

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Edward Lee, Chair
Professor Alberto Sangiovanni-Vincentelli
Professor Raja Sengupta

Fall 2006

The dissertation of James Adam Cataldo is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2006

The Power of Higher-Order Composition Languages in System Design

James Adam Cataldo

Abstract

The Power of Higher-Order Composition Languages in System Design

by

James Adam Cataldo

Doctor of Philosophy in Electrical Engineering

University of California, Berkeley

Professor Edward Lee, Chair

This dissertation shows the power of higher-order composition languages in system design. In order to formalize this, I develop an abstract syntax for composition languages and two calculi. The first calculus serves as a framework for composition languages without higher-order components. The second is a framework for higher-order composition languages. I prove there exist classes of systems whose specification in a higher-order composition language is drastically more succinct than it could ever be in a non-higher-order composition language.

To justify the calculus, I use it as a semantic domain for a simple higher-order composition language. I use it to reason about higher-order components in this more practical language and use n-level clock distribution networks as a class of systems whose description must grow exponentially in a non-higher order composition lan-

guage, but whose description grows linearly with n in a higher-order composition language.

As a prototype higher-order composition language, I developed the Ptalon programming language for Ptolemy II. I explain how components can be built in Ptalon, and I give several examples of models built as a higher-order components in this language. These examples span several domains in system design: control theory, signal processing, and distributed systems.

Unlike functional languages, where higher-order functions are infused with a program's execution semantics, the ability to provide scalable higher-order mechanism is completely separated from execution semantics in higher-order composition languages. As a design technique, higher-order composition languages can be used to provide extremely scalable system descriptions.

To Janie, queen of the obscure.

Contents

List of Figures

Acknowledgments

Of course, I must thank my mother and father for all the time, energy, money, etc. they have given me throughout my life. Everyone on my thesis committee has been very helpful, particularly Prof. Edward Lee, who has served as an inspiration for me during my time in grad school. I also owe him thanks for actually reading my dissertation multiple times. I must thank Prof. Alberto Sangiovanni-Vincentelli for inspiring some of the theoretical work in this dissertation and Prof. Raja Sengupta for giving me a great idea for an example problem.

Perhaps most importantly, I would like to thank Janie, who has been by my side offering support for the past five years. Without her constant jokes, my grad school experience would have been very dull.

For help on the Ptalon project, I must thank Thomas Feng, Elaine Cheong, and Andrew Mihal for their work on the original composition calculus. Jörn Janneck, Jie Liu, and Steve Neuendorffer have also given me some insight and ideas. I would also like to thank Jeffrey Dean for his feedback on my MapReduce example and Prof. Stephen Edwards for introducing me to higher-order components in SHIM.

For helping me on previous projects in grad school, I must thank Haiyang Zheng, Xiaojun Liu, and Eleftherios Matsikoudis for their work with me on abstract semantics. I must also thank Aaron Ames, Prof. Ian Mitchell, Prof. Shankar Sastry, Hoam Chung, Prof. Claire Tomlin, Prof. Karl Hedrik, and Prof. Pravin Varaiya for working with me on control theory. I'd also like to thank some other members of the Ptolemy

group with whom I've had the pleasure of many fruitful discussions over the years, especially Yang Zhao and Rachel Zhou, but also Christopher Hylands, James Yeh, Chris Chang, and Gang Zhou.

I can't forget to mention Dan Schonberg and Trevor Meyerowitz for all the good times at Berkeley. Finally, I must also thank the rest of my family–Cory, Mike, Jessica, Asia, Coral, and Lexie–for the support they've offered along the way.

Chapter 1

Introduction

Scalability is a big problem in system design. Kuetzer et al. [34] suggest an "orthogonalization of concerns" to help manage this problem. The idea is to separate various aspects of system design to allow more effective exploration of possible solutions in the design space. As an example, function, or what the system should do, can be separated from architecture, or how the system should do it. Similarly, communication, or how components transfer data, can be separated from computation, or how components transform data.

Another design methodology which addresses scale is Sztipanovits and Karsai's "model-integrated computing" [51]. Here the idea is to automate the transformation of a metamodel specification of a domain-specific modeling language, coupled with model integrity constraints, into a domain-specific design environment. Whenever possible, models, analysis tools, and transformation tools are shared across domains

to promote reuse and to save design time, both in a particular design environment and in the design of that design environment. Lédeczi et al. [1] identify an important orthogonalization of concerns in model-integrated computing, the separation of "static semantics," or the rules to specify which domain models are well-formed in a design environment, from "dynamic semantics," or the execution semantics of a system implementation or simulation.

Jackson and Sztipanovits [30] have taken this orthogonalization a step further by formalizing static and dynamic semantics in what they call "structural semantics" and "behavioral semantics" respectively. They formulate structural semantics as a decision procedure for checking model well-formedness in a particular domain. They formulate dynamic semantics as an "interpretation," or a mapping from one domain to another.

For any particular structural semantics, scalability can be an issue in the specification of system structure. Complex systems may require huge descriptions. A large system may contain a large amount of redundant structure, and exploiting this redundancy can lead to more succinct structural descriptions.

With this in mind, I propose *higher-order components* as a design technique to addresses the scalability concern. In functional programming languages, functions are referred to as "higher-order" because they may be stored in data structures, passed as arguments, and returned as results [27]. Similarly, in a *composition language*, or a language for constructing networks of components, the components are higher-

order when they may serve as parameters to other components. In a higher-order composition language, a system's structure is effectively parameterizable, and the parameters may be other systems.

As a design concern, the succinctness of a system description is orthogonal to its structural and behavioral well-formedness of models. Two expressions may represent the same system structure, but one may be much more compact than the other. The design goal of my technique is to minimize the amount of input a system designer must provide to create a new system, thus enabling one form of scalability in system design.

Unlike higher-order functions in a functional language, the semantics of higher-order components in a composition language are orthogonal to the execution semantics of the underlying model they represent. Given the parameters to a higher-order component in a model, the component may be "reduced" to a non-higher-order component before the system is simulated, verified, generated, or synthesized. This is a form of "partial evaluation." Partial evaluation is a program transformation technique for specializing programs with partial input known [13]. As an example, the Haskell function

```
f x = (2 + 3) * x
```

may be transformed to

```
f x = 5 * x
```

before the value of `x` is known. Executing the second program at some later time when

`x` is known, perhaps after the user supplies some input, requires less time, since the addition step has already been performed. In the context of higher-order composition languages, the designer must specify the parameters of a higher-order component before applying input data to the component in a simulation or synthesizing a circuit for the component. This is useful because the execution semantics, whether simulated, compiled, interpreted, or synthesized, need not be concerned with higher-order composition. The execution semantics need only be defined on the "flattened" model.

To show how higher-order components can lead to succinct syntactic descriptions of large systems, I formalize the notion of composition language in the form of a calculus. This calculus, which I call the "simple calculus," is general enough to represent system structures in a large class of design environments. The basic entities of the calculus are *ports*, or interface points through which data flows, and *relations*, which are combined with ports to provide connections. The basic operators of the calculus are *parallel composition*, which is used to combine entities in the calculus, *connection*, which is used to declare the routing of system data, and *encapsulation*, which is used to build system hierarchy.

In the calculus I present, I also include as basic entities *attributes*, which may be used to distinguish certain system characteristics at the structural level. Attributes are not completely fundamental to the calculus, since my main results would all hold without them. They serve a function in my calculus as similar to the function "constants" provide in λ calculus. Namely, application-specific information may be

encoded in the terms of the calculus using attributes. In λ calculus, the numbers $0, 1, 2, \ldots$ are commonly included in the calculus along with basic arithmetic operators to map practical functional languages, like Lisp and Haskell, to λ calculus. I will similarly use attributes when I map a higher-order circuit language to the calculus in Chapter 3. An example attribute might declare whether port is an input or output port. Since ports do not have directionality in all system environments, I would not want to include directionality as a primitive of the calculus. If I want to reason about the structural semantics of a model, I might want to reject models with input ports connected to input ports. Attributes afford me the ability to reason about such structural elements that go beyond the basic calculus.

In the simple calculus, there is no notion of a higher-order component. I extend this calculus with *variable abstraction* and *function application* to facilitate higher-order components. Along with this, I introduce a notion of *reduction* for terms in this "extended calculus." This reduction satisfies the *Church Rosser* property, which implies that each term in the calculus reduces to at most one term in the simple calculus. Equivalently, each higher-order component, when given parameters, reduces to at most one non-higher-order component.

I define a notion of *size* which is equivalent to the notion of size (or length) in λ calculus. The size of a term is roughly the number of symbols in the term. I prove that there exist classes of systems whose description in the extended calculus is drastically smaller than they could ever be in the simple calculus. This establishes

the expressive power that higher-order components can provide for succinct system descriptions in a composition language.

This succinctness comes with a cost, however. I define a notion of *structural equivalence*, which I can use to determine whether or not two terms in the calculus represent systems with the same structure, and a corresponding notion of *abstract syntax* for composition languages. Two terms in the simple calculus share the same abstract syntax if and only if they are structurally equivalent. In the simple calculus, I prove that checking structural equivalence of two models is always decidable. In the extended calculus, it is not. This need not be viewed as an overly negative result, because in practice it is often possible to determine structural equivalence of higher-order components. Moreover, this undecidability is a direct consequence of the expressiveness of the extended calculus.

To validate my calculus as a useful model, I develop a simple composition language for creating and combining Boolean circuits. To make the example interesting, the language permits higher-order components. I use the extended calculus as a semantic domain for this language. Using this mapping, I show a practical example of a class of systems whose size grows at least exponentially without using higher-order components, but whose size can be reduced to grow at most linearly when higher-order components are allowed.

As a prototype of this concept, I created Ptalon, a higher-order composition language that provides combinators for constructing higher-order components in Ptolemy

II. Ptolemy II is a platform for experimenting with new system design technologies for embedded systems [29]. Ptalon supports all of the features of higher-order components I mentioned above. Its semantics is orthogonal to the execution semantics of Ptolemy II since Ptalon uses partial evaluation to reduce higher-order components before a model is executed.

1.1 Background

Higher-order composition can be characterized as a design strategy in *actor-oriented design* [37] to address scalability. 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. Actor-oriented systems are commonly designed in a block-diagram environment. Lee [38] believes that actor-oriented design techniques will yield more reliable concurrent systems than those designed in a thread-based system. Examples of actor-oriented languages and design environments include hardware description languages (like VHDL [48] and Verilog[50]), coordination languages [47], architecture description languages [40], synchronous languages [7], Giotto [24], SystemC [3], SHIM [20], CAL [21], Simulink [17], LabVIEW [32], Metropolis [4], Ptolemy II [29], GME [36] and many more.

Ptolemy II, Metropolis, and GME deserve special attention as actor-oriented design environments for some of the novel design techniques they employ. Ptolemy

II supports a structured form of heterogeneous design called "hierarchical heterogeneity," where different concurrent interaction semantics may only be specified at different levels of the system hierarchy [22]. This structured form of heterogeneity simplifies analysis while still supporting multi-paradigm systems. Metropolis separates functional modeling from architecture modeling. "Quantity managers" capture the costs and constraints of the services an architecture provides in terms of various quantities, like time and energy. A "mapping network" maps the functionality onto the given architecture. In GME, users can design a new domain-specific modeling environment through metamodels, in addition to creating models in this environment. Model databases support the reuse of components across domains.

Both the desire for succinct descriptions of systems and the use of higher-order components are not new to system design. Methods for succinct descriptions of regular languages are well known. Meyer and Fischer [41] prove that a nondeterministic finite automaton with n states may require up to 2^n states if modeled as an equivalent deterministic finite automaton. Chandra et al. [11] introduce "alternating finite automata," an extension of nondeterministic finite automata, and prove that an alternating finite automaton with n states may require up to 2^{2^n} states to if modeled as an equivalent deterministic finite automaton. Drusinsky and Harel [19] go on to prove that an "alternating statechart" with n states may require up to $2^{2^{2^n}}$ states if modeled as an equivalent deterministic finite automaton. While these results are significant, not all systems can be modeled with regular languages, so these succinctness

results do not extend to all systems. For this reason, I base my succinctness results on the λ calculus notion of length [5], which I use to measure the size of terms in the extended calculus, rather than by counting the number of states in a state machine.

Given the obvious analogy between higher-order components and higher-order functions, it is probably no surprise that system design tools built on top of functional programming languages have supported higher-order components. The functional language Haskell [28] has been particularly influential in this regard. Reekie [49] layers Haskell on top of a dataflow process networks model to enable creation of higher-order system networks for real-time signal processing. The Lava Hardware Description Language [35], a Haskell extension targeted at hardware synthesis, supports higher-order circuits. Hawk [14] is an extension of Haskell for microprocessor simulation and verification, and also supports higher-order components. BlueSpec Classic [2], the original version of BlueSpec, has Haskell-like syntax and also supports higher-order components. Like Lava, BlueSpec is a hardware language, but in BlueSpec, systems are specified via cooperating finite state machines from which circuits are derived by the compiler rather than by a user manually specifying circuit functionality.

Outside of the Haskell world, higher-order components have been used as well, Janneck [31] implements higher-order Petri nets in the Moses system modeling tool to increase reusability of models. Ptolemy Classic and Ptolemy II [8] both support a limited set of higher-order components, but these components were built in the host language rather than in Ptolemy II itself. Ptalon, which is built on the Ptolemy

Expression Language, a purely functional language for manipulating Ptolemy-specific data types, is a higher-order composition language for composing networks of Ptolemy actors.

Many of the above examples use partial evaluation to reduce higher-order components to non-higher-order components. In Lava for instance, Haskell hardware specifications are reduced to non-higher-order components and transformed into VHDL. In Bluespec Classic, the specifications are transformed to the term-rewriting system, TRSpec [26], before being transformed to Verilog. Ptalon follows the same approach; higher-order components are flattened to non-higher-order Ptolemy models before execution.

1.2 Motivating Examples

The purpose of this section is to give two real-world examples of higher-order components. The first is a distributed systems example, and the second is a signal processing example. I have implemented both of these examples in Ptalon, and the screen shots are of the generated Ptolemy II components.

1.2.1 Map Reduce

The popular search engine Google uses huge amounts of data to provide users with information about everything from cell biology to basket weaving. Google maintains

tens of thousands of commodity machines to store and process its information. The Google File System (GFS) [23] is the distributed file system which helps manage these huge amounts of data.

Interacting directly with the GFS is too low-level for many of the data processing applications that software engineers at Google create. For this reason, the MapReduce [18] model of computation model was developed. The idea behind MapReduce is that users write their computation using two special methods, *map* and *reduce*. The map method takes a list of key-value pairs and outputs another list of key-value pairs. These returned lists are sorted by key, and the reduce function takes the list of all values associated with a particular key and returns a "reduced" list for that key. When computations are written this way, they can easily be parallelized, with map and reduce methods processing data on multiple machines. Dean [18] explains the full details of this distribution pattern.

An example application of MapReduce is a simple word count system. Here each input key is a web page, and each input value is a set of words on that web page. The goal is to count word frequency on an entire set of web pages. The map function takes in each key value pair; for each word on the page it emits that word as a key and the number of times it appears on the page as a value. The reduce function takes the list of counts for each key sums them. The output is the one-element list whose value represents the number of times that particular word appeared on any of the web pages.

I built a simulation of the MapReduce system in Ptolemy as a higher-order Ptalon component. The user of this component is prompted for the following parameters:

```
mapComponent
numberMaps
reduceComponent
numberReduces
```

The `mapComponent` and `reduceComoponent` are higher-order parameters. The `mapComponent` is responsible for the map computation and the `reduceComponent` is responsible for the reduce computation. I show the Ptalon code for this higher-order component below:

```
MapReduce is {
    /* The parameters for the mapreduce system.
     * The "fileName" parameter refers to the input file name.
     */
    actorparameter map;
    parameter numberOfMaps;
    actorparameter reduce;
    parameter numberOfReduces;
    parameter fileName;

    /* These are actor "constants" which tell Ptalon where to find
     * the "split" and "stop" actors.
     */
    actor split =
        ptalonActor:ptolemy.actor.ptalon.test.mapreduce.Split;
    actor stop =
        ptalonActor:ptolemy.actor.ptalon.test.mapreduce.WaitingStop;

    /* This output port is used to signal the status of the map reduce,
     * or whether the map reduce system has stopped computing its
     * results or not.
     */
    outport status;

    /* This creates the "split" actor and makes it ready for
```

```
 * connection. The split actor is responsible for distributing
 * data to the various map actors.
 */
port reference splitKeys;
port reference splitValues;
relation splitFinished;
split(keys := splitKeys, values := splitValues, doneReading :=
        splitFinished, file := [[fileName + ".map"]],
        numberOfOutputs := [[numberOfMaps]]);

/* This creates the "stop" actor, which is used to clean up file
 * handles after the computation has finished.
 */
port reference stopInput;
stop(input := stopInput, numberOfInputs := [[numberOfMaps]],
        status := status);

/* This creates the "reduce" actors and some connections.
 * This block of code will not be evaluated until the
 * parameters of this component have been set.
 */
for a initially [[0]] [[a < numberOfReduces]] {
    port reference reduceInKey[[a]];
    port reference reduceInValue[[a]];
    reduce(inputKey := reduceInKey[[a]], inputValue :=
            reduceInValue[[a]], doneReceiving := splitFinished,
            file := [[fileName + a.toString + ".red"]],
            numberOfInputs := [[numberOfMaps]]);
} next [[a + 1]]

/* This creates the "map" actors and some connections.
 * This block of code will not be evaluated until the
 * parameters of this component have been set.
 */
for a initially [[0]] [[a < numberOfMaps]] {
    port reference mapOutKeys[[a]];
    port reference mapOutValues[[a]];
    relation mapFinished[[a]];
    map(inputKey := splitKeys, inputValue := splitValues,
            outputKeys := mapOutKeys[[a]], outputValues :=
            mapOutValues[[a]], doneReceiving := splitFinished,
            doneEmitting := mapFinished[[a]],
```

```
                numberOfReduceOutputs := [[numberOfReduces]]);
        this(stopInput := mapFinished[[a]]);
        for b initially [[0]] [[b < numberOfReduces]] {
            this(reduceInKey[[b]] := mapOutKeys[[a]]);
            this(reduceInValue[[b]] := mapOutValues[[a]]);
        } next [[b + 1]]
    } next [[a + 1]]
}
```

In Figure 1.1, I show what the Ptolemy model corresponding to the MapReduce component for a particular set of input parameters. If I changed the number of map or reduce components, this topology would change correspondingly. The *Split* component is used to split the input file into key value pairs that are read in parallel by the *Map* components. The Map components apply the map function to each key value pair and emit a sequence of key value pairs. The *Reduce* components reduce lists of values for each key they receive. The *WaitingStop* component is used to signal the computation status to the outside world, and to make sure that model stops after all the data is processed. (If the model does not stop, the open file handles for the Split and Reduce components may cause memory leaks. The WaitingStop component is not part of the original MapReduce system, but the original system does manage the closing of file handles.)

To give some insight in the types of computations MapReduce is suited for, I show the basic map and reduce methods used by the map and reduce components for the word count system. Below that I show similar methods for computing the reverse hyperlink graph.

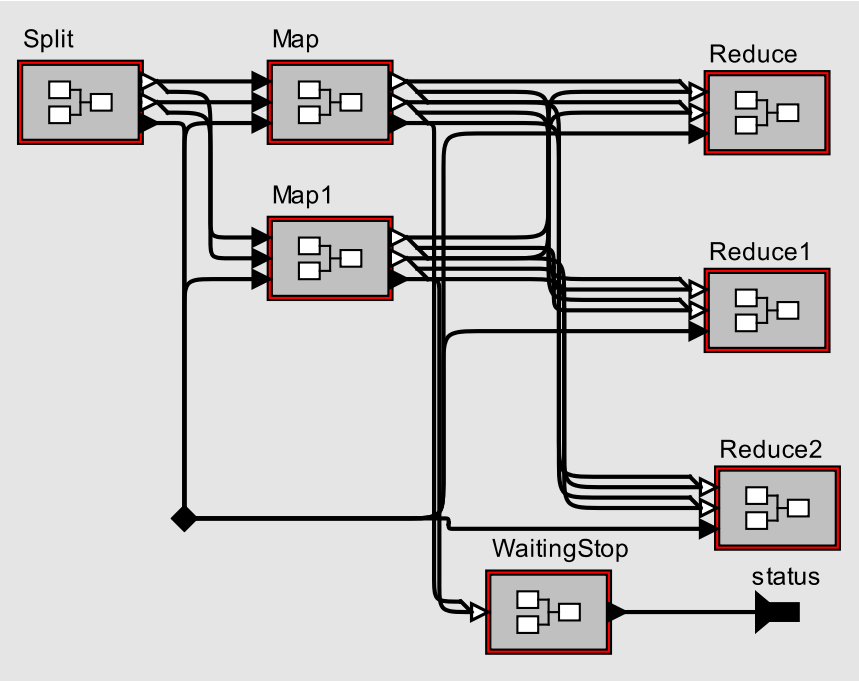

Figure 1.1: The Ptolemy model corresponding to the higher-order MapReduce component when the number of map components equals two and the number of reduce components equals three.

```
//Word count example
public List<KeyValuePair> map(String value) {
    //Each value is a list of words.
    //The output is a list of pairs, with the
    //first element being a word and the second element
    //the number of occurrences in the input string.
    StringTokenizer tokenizer = new StringTokenizer(value);
    LinkedList<KeyValuePair> output = new
        LinkedList<KeyValuePair>();
    while (tokenizer.hasMoreTokens()) {
        output.add(new KeyValuePair(tokenizer.nextToken(), "1"));
    }
    return output;
}

public List<String> reduce(String key, BlockingQueue<String>
        values) throws InterruptedException {
    //Each key is a word and a list of numbers comes in the
```

```
    //values Queue.  These numbers are added to compute
    //the total number of times the word was counted.
    int result = 0;
    while (!isQueueEmpty()) {
        String value = values.take();
        if (isQueueEmpty()) {
            break;
        }
        result += Integer.parseInt(value);
    }
    List<String> output = new LinkedList<String>();
    output.add((new Integer(result)).toString());
    return output;
}

////////////////////////////////////////////
//Reverse hyperlink example
    public List<KeyValuePair> map(String key, String value) {
        //Each input key is a web address and each value is
        //a web address the key links to.  The output
        //is simply the reversed pair.
        StringTokenizer tokenizer = new StringTokenizer(value);
        LinkedList<KeyValuePair> output =
                new LinkedList<KeyValuePair>();
        while (tokenizer.hasMoreTokens()) {
            output.add(new
                    KeyValuePair(tokenizer.nextToken(), key));
        }
        return output;
    }

    public List<String> reduce(String key, BlockingQueue<String>
            values) throws InterruptedException {
        //The input key is a web address and the values coming
        //in on the queue are the web addresses that link to
        //the key.  The linking web pages are assembled into
        //a list of all web pages that point to the key.
        List<String> output = new LinkedList<String>();
        while (!isQueueEmpty()) {
            String value = values.take();
            if (isQueueEmpty()) {
                break;
```

```
            }
            output.add(value);
        }
        return output;
    }
```

1.2.2 Fast Fourier Transform

Frequency analysis is one of the most basic tasks in signal processing. Lee and Varaiya [39] and Oppenheim et al. [46] both provide excellent introductions to the topic. A filter's frequency response, or how it amplifies and attenuates particular frequencies, is the main measure of its performance. The frequencies present in a signal are mathematically determined by an appropriate *Fourier transform*. While there are several flavors of Fourier transforms, only the *discrete Fourier transform* (DFT) can actually be numerically computed. Given a length N signal $x : \{0, 1, \ldots, N-1\} \to \mathbb{C}$, where $\mathbb{C}$ is the set of complex numbers, the DFT of x is defined as the function $X : \{0, 1, \ldots, N-1\} \to \mathbb{C}$, where for all $k \in \{0, 1, \ldots, N-1\}$,

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi n/N}$$

Computing the DFT using the brute force method requires $O(N^2)$ addition and multiplication computations. In 1965 Cooley and Tukey [15] popularized the *fast fourier transform* (FFT), which computes the DFT in only $O(N \log N)$ computations. There are many variants of FFTs. In the simplest case, N is a power of 2. Each N

point FFT is computed using two $N/2$ point FFTs. Oppenheim et al.[45] explains this algorithm in detail.

In Ptalon, it is straightforward to make an FFT as a higher-order Ptolemy component. This is a classic higher-order component. For instance, Claesson et al. [35] cite it as one instance of a "butterfly circuit." I show the code for this higher-order component below:

```
FFT is {
    parameter N;
    actor gain = ptolemy.actor.lib.Scale;
    actor adder = ptolemy.actor.ptalon.demo.ComplexAddSubtract;
    if [[ (N >= 2) && ((N % 2) == 0) ]] {
        inport[] x;
        outport[] X;
        if [[ N == 2 ]] {
            relation x0;
            relation x1;
            this(x := x0);
            this(x := x1);
            port reference sum0;
            port reference sum1;
            adder(plus := sum0, output := X);
            adder(plus := sum1, output := X);
            this(sum0 := x0);
            this(sum1 := x0);
            gain(input := x1, output := sum0, factor := [[ 1 ]]);
            gain(input := x1, output := sum1, factor := [[ -1 ]]);
        } else {
            port reference xFromInput;
            this(x := xFromInput);
            port reference xEven;
            port reference xOdd;
            port reference G;
            port reference H;
            FFT(x := xEven, X := G, N := [[N / 2]]);
            FFT(x := xOdd, X := H, N := [[N / 2]]);
```

```
            for n initially [[0]] [[n < N / 2]] {
                relation G[[n]];
                relation H[[n]];
                this(G := G[[n]], H := H[[n]]);
            } next [[n + 1]]

            for n initially [[0]] [[n < N]] {
                relation x[[n]];
                this(xFromInput := x[[n]]);
                if [[ (n % 2) == 0 ]] {
                    this(xEven := x[[n]]);
                } else {
                    this(xOdd := x[[n]]);
                }

                port reference sum[[n]];
                adder(plus := sum[[n]], output := X);

                if [[ n < N / 2]] {
                    gain(input := H[[n]], output := sum[[n]],
                            factor :=[[ exp(-j*2*n*pi/N) ]]);
                    this(sum[[n]] := G[[n]]);
                } else {
                    gain(input := H[[n - N/2]], output := sum[[n]],
                            factor :=[[ exp(-j*2*n*pi/N) ]]);
                    this(sum[[n]] := G[[n - N/2]]);
                }
            } next [[n + 1]]
        }
    } else {
        /* Do nothing if N is not a power of 2. */
    }
}
```

In figure 1.2, I show the Ptolemy model generated from the FFT component when N is 4. Note that the FFT has a recursive structure. Recursion can be viewed as a special type of higher-order composition, where a composition is parameterized by itself.

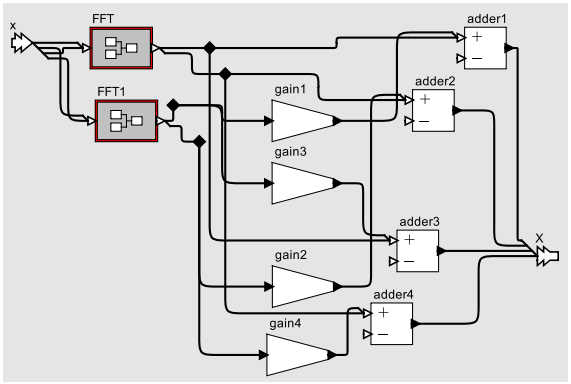

Figure 1.2: The Ptolemy model corresponding to a decimation-in-time FFT higher-order component when $N = 4$.

Chapter 2

The Succinctness of Higher-Order Composition Languages

In this chapter, I provide a mathematical framework for higher-order composition languages. I start with a *simple calculus* that serves as a framework for non-higher-order composition languages. I then extend this with the *extended calculus*, which serves as a framework for higher-order composition languages.

An important result states that I can express models more succinctly in the extended calculus than in the simple calculus. What's startling about this result is the degree to which this can be done. I can find a countably infinite set of terms in the simple calculus whose size grows at least as fast as 2^n, and for which no other terms in the simple calculus can be smaller. Using the extended calculus, I can find equivalent higher-order terms whose size grows only linearly. The result doesn't

stop there. I can find another countably infinite set of terms in the simple calculus whose size grows at least as fast as 2^{2^n}, and for which no other terms in the simple calculus can be smaller. I can still find equivalent terms in the extended calculus whose size grows only linearly. The result keeps going forever. That is, I can find a countably infinite set of terms in the simple calculus whose size grows at least as fast as $\text{pow}(2, \text{pow}(2, \ldots \text{pow}(2, n) \cdots))$, for an arbitrary number of exponentiations, and for which no other terms in the simple calculus can be smaller, yet I can still find equivalent terms in the extended calculus whose size grows only linearly.

I note that I had originally developed another calculus which also serves as a framework for higher-order composition languages in [10]. This previous calculus is much smaller and simpler than the extended calculus I develop here. While the earlier calculus is effective in capturing the essence of higher-order composition languages, there is no way for me to reason about non-higher-order components, so I developed this more elaborate model.

2.1 The Simple Calculus

Consider a simple component which has two ports, named p_1 and p_2 and whose name is l_1. In the simple calculus, this can be described as

$$l_1 : [p_1 \oplus p_2]$$

Here l_1 serves as a label for the component. A picture for this structure is shown in Figure 2.1.

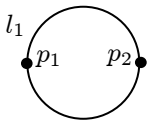

Figure 2.1: A two port component in the simple calculus

The picture in Figure 2.1 is similar in both character and spirit to Milner's flowgraphs [42]. Milner's flowgraphs are visual representations of the *static* combinators in his calculus of communicating systems (CCS). These combinators deal with the interconnection structure of components, as does this calculus. The reason for inventing a new calculus rather than using static CCS as a starting point is simply that this new calculus will be easier to extend to the higher-order calculus of the next section.

A script letter, like $\mathcal{A}$, possibly with a subscript, abbreviates a term in the calculus. If $\mathcal{A}$ and $\mathcal{B}$ represent the same expression, I write

$$\mathcal{A} \equiv \mathcal{B}$$

This means $\mathcal{A}$ and $\mathcal{B}$ are *syntactically equivalent*. When I wish to define $\mathcal{A}$ to be the syntactic equivalent of some other expression, I use the $\equiv_{def}$ symbol, like

$$\mathcal{A} \equiv_{def} l_1 : [p_1]$$

Note that ordering is somewhat irrelevant in this calculus. If I define $\mathcal{A}$ by

$$\mathcal{A} \equiv_{def} l_1 : [p_1 \oplus p_2]$$

and $\mathcal{B}$ by

$$\mathcal{B} \equiv_{def} l_1 : [p_2 \oplus p_1]$$

Then $\mathcal{A} =_{\mathfrak{S}} \mathcal{B}$, where $=_{\mathfrak{S}}$ is the symbol for *structural equivalence*. Note that $\mathcal{A} \not\equiv \mathcal{B}$, that is they are not syntactically equivalent, but they are structurally equivalent, which indicates that they both represent the same system.

While the order is secondary in this calculus, the names of ports are important. It is not the case that

$$l_1 : [p_1 \oplus p_2] =_{\mathfrak{S}} l_1 : [p_3 \oplus p_4]$$

Some elements included in particular composition languages structure may not be captured using just ports and relations. For instance, the calculus says nothing about the data types of ports or whether a port is an input or output. For such situations, I include *attributes* in the calculus.

That said, it can be useful to distinguish two components in this calculus that have the same port names. They may represent very different systems. For instance, in a practical system we may have many different types of components which all have ports named *input* and *output*. We use *attributes* to make such distinctions. For the same component with an attribute A_1, I write

$$l_1 : [A_1 \oplus p_1 \oplus p_2]$$

Consider the term

$$\mathcal{T} \equiv_{def} A_1 \oplus p_1$$

Suppose I wish to form a network of two $\mathcal{T}$ components in parallel. I can use the following expression:

$$\mathcal{S} \equiv_{def} l_1 : [\mathcal{T}] \oplus l_2 : [\mathcal{T}] \tag{2.1}$$

which means that

$$\mathcal{S} \equiv l_1 : [A_1 \oplus p_1] \oplus l_2 : [A_1 \oplus p_1]$$

The *component labels* then simply give me a way to distinguish two instances of the same component. This can be seen in Figure 2.2.

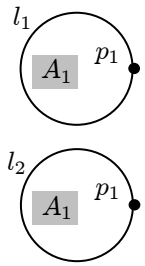

Figure 2.2: A network of two components in parallel.

I can now transform the network $\mathcal{S}$ of Equation 2.1 into a new component using hierarchy. To do this, I let

$$\begin{aligned} \mathcal{U} \quad &\equiv_{def} \quad l_3 : [\mathcal{S} \oplus p_1 \oplus p_2 \oplus l_1\{p_1 := p_1\} \oplus l_2\{p_1 := p_2\}] \\ &\equiv \quad l_3 : [l_1 : [A_1 \oplus p_1] \oplus l_2 : [A_1 \oplus p_1] \oplus p_1 \oplus p_2 \oplus l_1\{p_1 := p_1\} \oplus l_2\{p_1 := p_2\}] \end{aligned}$$

This gives the component in Figure 2.3. Note that the term $l_2\{p_1 := p_2\}$ means to connect the port p_1 of the component l_2 to port p_2 of its container.

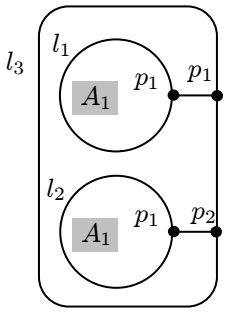

Figure 2.3: A component built out of a parallel network.

Now consider a new term named $\mathcal{T}$:

$$\mathcal{T} \equiv_{def} A_1 \oplus p_1 \oplus p_2$$

Suppose, I wish to create a new component out of two copies of $\mathcal{T}$ in series. I can use

$$l_3 : [p_1 \oplus p_2 \oplus r_1 \oplus l_1 : [\mathcal{T}] \oplus l_2 : [\mathcal{T}] \oplus l_1\{p_1 := p_1\} \oplus \alpha\{p_2 := r_1\} \oplus$$
$$l_2\{p_1 := r_1\} \oplus l_2\{p_2 := p_2\}]$$

This is shown in figure 2.4. Here r_1 is a *relation*, or a connection point which does not propagate to the interface of the component it belongs to.[1] A relation is like a "wire" in Verilog [50].

2.1.1 Formal Details of the Simple Calculus

In this section, I provide the formal details of the simple calculus. I define its syntax and structural equivalence, and I conclude with a proof that for all terms $\mathcal{A}$ and $\mathcal{B}$ in the simple calculus, the proposition $\mathcal{A} =_{\Theta} \mathcal{B}$ is decidable.

[1] π calculus [44] uses the name binding operator to convert ports to relations.

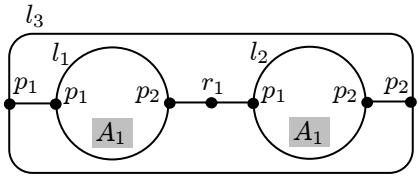

Figure 2.4: A series component in the calculus.

Let $\mathfrak{P}$ be the set of all port names, $\mathfrak{R}$ the set of relation names, $\mathfrak{A}$ the set of all attribute names, and $\mathfrak{L}$ the set of all component label names. I assume all four sets are mutually disjoint and denumerable and use the naming conventions above, so that $p_0, p_1, \ldots \in \mathfrak{P}$, $r_0, r_1, \ldots \in \mathfrak{R}$, $A_0, A_1, \ldots \in \mathfrak{A}$, and $l_0, l_1, \ldots \in \mathfrak{L}$.

Notice that all names have a natural number as a subscript. To define the calculus, I need a way to refer to an arbitrary name. I use the *name variables* i, j, k, a, b, and c for this purpose. By convention, p_2 is a port name and p_j represents an arbitrary port name, but p_j is not itself a port name. If I say $p_j \in \mathfrak{P}$, I am referring to an arbitrary element of $\mathfrak{P}$, but if I say $p_2 \in \mathfrak{P}$ I'm referring to the specific element p_2 of $\mathfrak{P}$. The same convention holds for attribute names, relation names, and component label names. I will always assume that name variables take values in the natural numbers. If I assert that $i = j$, I am asserting that they are variables that represent the same natural number, which would imply for instance that $r_i \equiv r_j$.

I define the set $\mathfrak{S}$ of terms in the simple calculus to be the smallest set such that

$$
\begin{aligned}
true \Rightarrow \quad & 0 \in \mathfrak{S} \\
p_i \in \mathfrak{P} \Rightarrow \quad & p_i \in \mathfrak{S} \\
r_i \in \mathfrak{R} \Rightarrow \quad & r_i \in \mathfrak{S} \\
A_i \in \mathfrak{A} \Rightarrow \quad & A_i \in \mathfrak{S} \\
l_i \in \mathfrak{L} \ \wedge \ p_j, p_k \in \mathfrak{P} \Rightarrow \quad & l_i\{p_j := p_k\} \in \mathfrak{S} \\
l_i \in \mathfrak{L} \ \wedge \ p_j, \in \mathfrak{P} \ \wedge \ r_k \in \mathfrak{R} \Rightarrow \quad & l_i\{p_j := r_k\} \in \mathfrak{S} \\
\mathcal{T} \in \mathfrak{S} \ \wedge \ \mathcal{S} \in \mathfrak{S} \Rightarrow \quad & (\mathcal{T} \oplus \mathcal{S}) \in \mathfrak{S} \\
\mathcal{T} \in \mathfrak{S} \ \wedge \ l_i \in \mathfrak{L} \Rightarrow \quad & l_i{:}[\mathcal{T}] \in \mathfrak{S}
\end{aligned}
$$

Here 0 is the *empty model*. I include it here to have a notion of nothing. It will be mainly used when I extend this calculus to the higher-order calculus, but since it is not a higher-order term, I include it in the simple calculus.

I now define the structural equivalence relation $=_{\mathfrak{S}}$ as the minimal subset of $\mathfrak{S} \times \mathfrak{S}$

such that for all $\mathcal{T}, \mathcal{S}, \mathcal{U} \in \mathfrak{S}$,

$$\begin{aligned}
\mathcal{T} \equiv \mathcal{S} \Rightarrow \quad & \mathcal{T} =_{\mathfrak{S}} \mathcal{S} \\
true \Rightarrow \quad & (\mathcal{T} \oplus \mathcal{S}) =_{\mathfrak{S}} (\mathcal{S} \oplus \mathcal{T}) \\
true \Rightarrow \quad & ((\mathcal{T} \oplus \mathcal{S}) \oplus \mathcal{U}) =_{\mathfrak{S}} (\mathcal{T} \oplus (\mathcal{S} \oplus \mathcal{U})) \\
true \Rightarrow \quad & (\mathcal{T} \oplus 0) =_{\mathfrak{S}} \mathcal{T} \\
i = j \Rightarrow \quad & (p_i \oplus p_j) =_{\mathfrak{S}} p_i \\
i = j \Rightarrow \quad & (r_i \oplus r_j) =_{\mathfrak{S}} r_i \\
i = j \Rightarrow \quad & (A_i \oplus A_j) =_{\mathfrak{S}} A_i \\
i = a \ \wedge \ j = b \ \wedge \ k = c \Rightarrow \quad & (l_i\{p_j := p_k\} \oplus l_a\{p_a := p_b\}) =_{\mathfrak{S}} l_i\{p_j := p_k\} \\
i = a \ \wedge \ j = b \ \wedge \ k = c \Rightarrow \quad & (l_i\{p_j := r_k\} \oplus l_a\{p_a := r_b\}) =_{\mathfrak{S}} l_i\{p_j := r_k\} \\
i = j \Rightarrow \quad & (l_i\!:\![\mathcal{T}] \oplus l_j\!:\![\mathcal{S}]) =_{\mathfrak{S}} l_i\!:\![(\mathcal{T} \oplus \mathcal{S})] \\
\mathcal{T} =_{\mathfrak{S}} \mathcal{S} \Rightarrow \quad & l_i\!:\![\mathcal{T}] =_{\mathfrak{S}} l_i\!:\![\mathcal{S}] \\
\mathcal{T} =_{\mathfrak{S}} \mathcal{S} \Rightarrow \quad & \mathcal{S} =_{\mathfrak{S}} \mathcal{T} \\
\mathcal{T} =_{\mathfrak{S}} \mathcal{S} \ \wedge \ \mathcal{S} =_{\mathfrak{S}} \mathcal{U} \Rightarrow \quad & \mathcal{T} =_{\mathfrak{S}} \mathcal{U}
\end{aligned}$$

To put these rules in words, "duplicates don't matter", and "order doesn't matter". If a term has two ports named p_i, rather than considering this to be an "error" term, it is simply a term with redundant information. Because of the associativity and commutivity of $\oplus$, I will omit parenthesis in further expressions in the simple calculus. I now have the main result of this section, namely that it is always possible

to determine whether two terms in the simple calculus are equivalent. By the finite nature of terms, it is somewhat trivial, but I include the theorem to contrast the simple calculus with the extended calculus. The result does not hold in the extended calculus.

Theorem 2.1. *For all $\mathcal{S}, \mathcal{T} \in \mathfrak{S}$, the proposition $\mathcal{S} =_{\mathfrak{S}} \mathcal{T}$ is decidable.*

Proof. The following test, based on the structure of $\mathcal{S}$ and $\mathcal{T}$ will always determine the truth of the proposition:

1. **Case $\mathcal{S} \equiv 0$.** Suppose $\mathcal{T} \equiv 0$. Then $\mathcal{S} =_{\mathfrak{S}} \mathcal{T}$ is true. Suppose $\mathcal{T}$ is one of p_i, r_i, A_i, $l_i\{p_j := p_k\}$, $l_i\{p_j := r_k\}$, or $l_i : [\mathcal{U}]$ for some expression $\mathcal{U}$. Then the proposition is false. Finally suppose $\mathcal{T} \equiv \mathcal{T}_1 \oplus \mathcal{T}_2$ for some expressions $\mathcal{T}_1$ and $\mathcal{T}_2$. If $0 =_{\mathfrak{S}} \mathcal{T}_1$ and $0 =_{\mathfrak{S}} \mathcal{T}_2$, then the proposition is true, otherwise it is false.

2. **Case $\mathcal{S} \equiv p_i$.** Suppose $\mathcal{T} \equiv p_j$. Then $\mathcal{S} =_{\mathfrak{S}} \mathcal{T}$ is true if and only if $p_i \equiv p_j$. Suppose $\mathcal{T}$ is one of 0, r_i, A_i, $l_i\{p_j := p_k\}$, $l_i\{p_j := r_k\}$, or $l_i : [\mathcal{U}]$ for some expression $\mathcal{U}$. Then the proposition is false. Finally suppose $\mathcal{T} \equiv \mathcal{T}_1 \oplus \mathcal{T}_2$ for some expressions $\mathcal{T}_1$ and $\mathcal{T}_2$. If $p_i =_{\mathfrak{S}} \mathcal{T}_1$ and $p_i =_{\mathfrak{S}} \mathcal{T}_2$, then the proposition is true, otherwise it is false.

3. **Case $\mathcal{S} \equiv r_i$.** Suppose $\mathcal{T} \equiv r_j$. Then $\mathcal{S} =_{\mathfrak{S}} \mathcal{T}$ is true if and only if $r_i \equiv r_j$. Suppose $\mathcal{T}$ is one of 0, p_i, A_i, $l_i\{p_j := p_k\}$, $l_i\{p_j := r_k\}$, or $l_i : [\mathcal{U}]$ for some expression $\mathcal{U}$. Then the proposition is false. Finally suppose $\mathcal{T} \equiv \mathcal{T}_1 \oplus \mathcal{T}_2$ for

some expressions $\mathcal{T}_1$ and $\mathcal{T}_2$. If $r_i =_{\mathfrak{S}} \mathcal{T}_1$ and $r_i =_{\mathfrak{S}} \mathcal{T}_2$, then the proposition is true, otherwise it is false.

4. **Case $\mathcal{S} \equiv A_i$.** Suppose $\mathcal{T} \equiv A_j$. Then $\mathcal{S} =_{\mathfrak{S}} \mathcal{T}$ is true if and only if $A_i \equiv A_j$. Suppose $\mathcal{T}$ is one of 0, p_i, r_i, $l_i\{p_j := p_k\}$, $l_i\{p_j := r_k\}$, or $l_i : [\mathcal{U}]$ for some expression $\mathcal{U}$. Then the proposition is false. Finally suppose $\mathcal{T} \equiv \mathcal{T}_1 \oplus \mathcal{T}_2$ for some expressions $\mathcal{T}_1$ and $\mathcal{T}_2$. If $A_i =_{\mathfrak{S}} \mathcal{T}_1$ and $A_i =_{\mathfrak{S}} \mathcal{T}_2$, then the proposition is true, otherwise it is false.

5. **Case $\mathcal{S} \equiv l_i\{p_j := p_k\}$.** Suppose $\mathcal{T} \equiv l_a\{p_b := p_c\}$. Then $\mathcal{S} =_{\mathfrak{S}} \mathcal{T}$ is true if and only if $l_i \equiv l_a$, $p_j \equiv p_b$, and $p_k \equiv p_c$. Suppose $\mathcal{T}$ is one of 0, p_i, r_i, A_i, $l_i\{p_j := r_k\}$, or $l_i : [\mathcal{U}]$ for some expression $\mathcal{U}$. Then the proposition is false. Finally suppose $\mathcal{T} \equiv \mathcal{T}_1 \oplus \mathcal{T}_2$ for some expressions $\mathcal{T}_1$ and $\mathcal{T}_2$. If $l_i\{p_j := p_k\} =_{\mathfrak{S}} \mathcal{T}_1$ and $l_i\{p_j := p_k\} =_{\mathfrak{S}} \mathcal{T}_2$, then the proposition is true, otherwise it is false.

6. **Case $\mathcal{S} \equiv l_i\{p_j := r_k\}$.** Suppose $\mathcal{T} \equiv l_a\{p_b := r_c\}$. Then $\mathcal{S} =_{\mathfrak{S}} \mathcal{T}$ is true if and only if $l_i \equiv l_a$, $p_j \equiv p_b$, and $r_k \equiv r_c$. Suppose $\mathcal{T}$ is one of 0, p_i, r_i, A_i, $l_i\{p_j := p_k\}$, or $l_i : [\mathcal{U}]$ for some expression $\mathcal{U}$. Then the proposition is false. Finally suppose $\mathcal{T} \equiv \mathcal{T}_1 \oplus \mathcal{T}_2$ for some expressions $\mathcal{T}_1$ and $\mathcal{T}_2$. If $l_i\{p_j := r_k\} =_{\mathfrak{S}} \mathcal{T}_1$ and $l_i\{p_j := r_k\} =_{\mathfrak{S}} \mathcal{T}_2$, then the proposition is true, otherwise it is false.

7. **Case $\mathcal{S} \equiv l_i : [\mathcal{S}_0]$, for some $\mathcal{S}_0 \in \mathfrak{S}$.** Suppose $\mathcal{T} \equiv l_j : [\mathcal{T}_0]$, for some $\mathcal{T}_0 \in \mathfrak{S}$. Then $\mathcal{S} =_{\mathfrak{S}} \mathcal{T}$ is true if and only if $l_i \equiv l_j$ and $\mathcal{S}_0 =_{\mathfrak{S}} \mathcal{T}_0$. Suppose $\mathcal{T}$ is one of 0, p_i, r_i, A_i, $l_i\{p_j := p_k\}$, or $l_i\{p_j := r_k\}$. Then the proposition is false.

Finally suppose $\mathcal{T} \equiv \mathcal{T}_1 \oplus \mathcal{T}_2$ for some expressions $\mathcal{T}_1$ and $\mathcal{T}_2$. If $\mathcal{T}_1 \equiv l_j : [\mathcal{T}_3]$, $\mathcal{T}_2 \equiv l_k : [\mathcal{T}_4]$, $l_i \equiv l_j \equiv l_k$, and $\mathcal{S}_0 =_{\mathfrak{S}} \mathcal{T}_3 \oplus \mathcal{T}_4$, then the proposition is true, otherwise it is false.

8. **Case $\mathcal{S} \equiv \mathcal{S}_1 \oplus \mathcal{S}_2$, for some $\mathcal{S}_1, \mathcal{S}_2 \in \mathfrak{S}$.** Suppose $\mathcal{T}$ is one of 0, p_i, r_i, A_i, $l_i\{p_j := p_k\}$, or $l_i\{p_j := r_k\}$, $l_i : [\mathcal{U}]$ for some expression $\mathcal{U}$. Then the proposition is true if and only if $\mathcal{T} =_{\mathfrak{S}} \mathcal{S}$, which we can test with one of the above cases. Finally suppose $\mathcal{T} \equiv \mathcal{T}_1 \oplus \mathcal{T}_2$ for some expressions $\mathcal{T}_1$ and $\mathcal{T}_2$. There exists some positive integers a and b such that $\mathcal{S} \equiv \mathcal{S}_1 \oplus \cdots \oplus \mathcal{S}_a$ and $\mathcal{T} \equiv \mathcal{T}_1 \oplus \cdots \oplus \mathcal{T}_b$ and each $\mathcal{S}_c$ and $\mathcal{T}_d$ is one of 0, p_i, r_i, A_i, $l_i\{p_j := p_k\}$, or $l_i\{p_j := r_k\}$, $l_i : [\mathcal{U}]$ for some expression $\mathcal{U}$. Then $\mathcal{S} =_{\mathfrak{S}} \mathcal{T}$ is true if and only if (a) for each $\mathcal{S}_c \in \{\mathcal{S}_1, \ldots, \mathcal{S}_a\}$ there is a $\mathcal{T}_d \in \{\mathcal{T}_1, \ldots, \mathcal{T}_b\}$ such that $\mathcal{S}_c =_{\mathfrak{S}} \mathcal{T}_d$, and (b) for each $\mathcal{T}_d \in \{\mathcal{T}_1, \ldots, \mathcal{T}_b\}$ there is a $\mathcal{S}_c \in \{\mathcal{S}_1, \ldots, \mathcal{S}_a\}$ such that $\mathcal{S}_c =_{\mathfrak{S}} \mathcal{T}_d$. Each of these tests can also be carried out with one of the above tests.

The tests above always terminate with the logical value of the proposition, so I conclude that the proposition is decidable. □

2.1.2 Structural Equivalence and Abstract Syntax

Note that the simple calculus provides a concrete syntax for composition languages. Structural equivalence gives a way to test if two terms in the calculus represent the same system. In this section, I provide an abstract syntax for composition languages. Two terms will be structurally equivalent if and only if they have the same

abstract syntax. Define the set Syn as the smallest set such that

$$\begin{aligned}
true \Rightarrow \quad & \emptyset \in Syn \\
p_i \in \mathfrak{P} \Rightarrow \quad & \{p_i\} \in Syn \\
r_i \in \mathfrak{R} \Rightarrow \quad & \{r_i\} \in Syn \\
A_i \in \mathfrak{A} \Rightarrow \quad & \{A_i\} \in Syn \\
l_i \in \mathfrak{L} \ \wedge \ p_j, p_k \in \mathfrak{P} \Rightarrow \quad & \big\{(p_j, p_k, l_i)\big\} \in Syn \\
l_i \in \mathfrak{L} \ \wedge \ p_j \in \mathfrak{P} \ \wedge \ r_k \in \mathfrak{R} \Rightarrow \quad & \big\{(p_j, r_k, l_i)\big\} \in Syn \\
S_i, S_j \in Syn \Rightarrow \quad & S_i \cup S_j \in Syn \\
S \in Syn, l_i \in \mathfrak{L} \Rightarrow \quad & \big\{(l_i, S_j) \big| S_j \in S\big\} \in Syn
\end{aligned}$$

Any element of Syn will be an *abstract term*.

Define the function $Ab : \mathfrak{S} \to Syn$ inductively as follows:

$$\begin{aligned}
Ab[\![0]\!] &= \{0\} \\
Ab[\![p_i]\!] &= \{p_i\} \\
Ab[\![r_i]\!] &= \{r_i\} \\
Ab[\![A_i]\!] &= \{A_i\} \\
Ab[\![l_i\{p_j := p_k\}]\!] &= \{(p_j, p_k, l_i)\} \\
Ab[\![l_i\{p_j := r_k\}]\!] &= \{(p_j, r_k, l_i)\} \\
Ab[\![\mathcal{T} \oplus \mathcal{S}]\!] &= Ab[\![\mathcal{T}]\!] \cup Ab[\![\mathcal{S}]\!] \\
Ab[\![l_i : [\mathcal{T}]]\!] &= \big\{(l_i, S_j) \big| S_j \in Ab[\![\mathcal{T}]\!]\big\}
\end{aligned}$$

The function Ab provides an abstract syntax for $\mathfrak{S}$ by providing an abstract term for each concrete term in $\mathfrak{S}$. Note the following theorem:

Theorem 2.2. *For all* $\mathcal{T}, \mathcal{S} \in \mathfrak{S}$, $\mathcal{T} =_{\mathfrak{S}} \mathcal{S}$ *if and only if* $Ab[\![\mathcal{T}]\!] = Ab[\![\mathcal{S}]\!]$.

This result is immediate from the definition of $=_S$ and Ab, so I do not prove it here.

2.2 The Higher-Order Calculus

From this point forward, I will assume the reader has at least a basic familiarity with λ calculus. To the reader who has yet to learn this elegant formalism, Barendregt [6] provides a gentle introduction. Hindley [25] provides more details for the intermediate learner, and Barendregt [5] provides a more complete exposition of λ calculus more appropriate to a "λ expert."

Imagine creating a new network by combining two instances of *any* component in parallel. Maybe the component design is incomplete, or maybe several possibilities could fit into the same pattern. A *variable* in the extended calculus can represent such a component. Suppose I use x_1 to represent this variable. (Note that x_1 is a variable name in the extended calculus, not a variable used to represent an arbitrary name in the calculus.) Then the expression:

$$\lambda x_1.l_1 : [x_1] \oplus l_2 : [x_1]$$

will do the job. Such a term is called an *abstraction*, where the parameter x_1 is abstracted from the system description. This now represents a parameterizable structure, or a higher-order component. Just as in λ calculus, I may give another expression as a parameter to this expression. For example the term

$$\big(\big(\lambda x_1.l_1 : [x_1] \oplus l_2 : [x_1]\big) \;\; (A_1 \oplus p_1 \oplus p_2) \big)$$

reduces to the term

$$l_1 : [A_1 \oplus p_1 \oplus p_2] \oplus l_2 : [A_1 \oplus p_1 \oplus p_2]$$

This is simply a *function application.*

Using just variables, abstractions, and applications, I can create the ordinary λ calculus as a subset of the extended calculus. I can then define Church numerals:

$$\begin{aligned} \mathcal{C}_0 &\equiv_{def} \lambda x_1.\lambda x_2.x_2 \\ \mathcal{C}_{n+1} &\equiv_{def} (\lambda x_1.\lambda x_2.\lambda x_1.(x_2\ ((x_1\ x_2)\ x_1)))\ \mathcal{C}_n \end{aligned}$$

where each $\mathcal{C}_i$ represents the natural number i. I can similarly define the boolean values:

$$\begin{aligned} \mathcal{B}_{true} &\equiv_{def} \lambda x_1.\lambda x_2.x_1 \\ \mathcal{B}_{false} &\equiv_{def} \lambda x_1.\lambda x_2.x_2 \end{aligned}$$

where $\mathcal{B}_{true}$ represents logical true and $\mathcal{B}_{false}$ represent logical false. The Church-Turing thesis states that any computation can be encoded in λ calculus. Two such

computations that I will find repeatedly useful are the predecessor and zero functions:

$$\mathcal{F}_{pred} \equiv_{def} \lambda x_1.\lambda x_2.\lambda x_3.(x_1\ (\lambda x_4.\lambda x_5.(x_5\ (x_4\ x_2)))) \tag{2.2}$$

$$\mathcal{F}_{zero} \equiv_{def} \lambda x_1.(x_1\ ((\mathcal{B}_{true}\ \mathcal{B}_{false})\ \mathcal{B}_{true})) \tag{2.3}$$

It can be checked that for all natural numbers i,

$$(\mathcal{F}_{pred}\ \mathcal{C}_{i+1}) \twoheadrightarrow_\beta \mathcal{C}_i$$

$$(\mathcal{F}_{zero}\ \mathcal{C}_{i+1}) \twoheadrightarrow_\beta \mathcal{B}_{false}$$

$$(\mathcal{F}_{zero}\ \mathcal{C}_0) \twoheadrightarrow_\beta \mathcal{B}_{true}$$

where $\mathcal{T} \twoheadrightarrow_\beta \mathcal{S}$ means $\mathcal{T}$ *reduces to* $\mathcal{S}$. In order to have recursion, I need a fixed-point combinator:

$$\mathcal{Y}_{fp} \equiv_{def} ((\lambda x_1.\lambda x_2.(x_2\ ((x_1\ x_1)\ x_2)))\ (\lambda x_1.\lambda x_2.(x_2\ ((x_1\ x_1)\ x_2))))$$

This is Turing's combinator, and it has the nice property that for any expression $\mathcal{T}$,

$$(\mathcal{Y}_{fp}\ \mathcal{T}) \twoheadrightarrow_\beta (\mathcal{T}\ (\mathcal{Y}_{fp}\ \mathcal{T})) \tag{2.4}$$

Using this number system, I can define a higher-order component that represents x_3 copies of component x_2 in parallel:

$$\mathcal{T}_{parallel} \equiv_{def} (\mathcal{Y}_{fp}\ \ \lambda x_1.\lambda x_2.\lambda x_3.((\mathcal{F}_{zero}\ x_3)\ 0\ (l_{\langle x_3 \rangle} : [x_2] \oplus (x_1\ x_2\ (\mathcal{F}_{pred}\ x_3))))\ )$$

For all $i > 0$,

$$((\mathcal{T}_{parallel}\ \mathcal{S})\ \mathcal{C}_i)$$

reduces to a term in the simple calculus with i copies of of $\mathcal{S}$ in parallel. For instance

$$((\mathcal{T}_{parallel}\ A_1)\ \mathcal{C}_3)$$

reduces to

$$l_1 : [A_1] \oplus l_2 : [A_1] \oplus l_3 : [A_1]$$

Similarly, suppose I wish to create a component with x_2 ports, named $p_1, p_2, \ldots, p_{x_2}$. I can use

$$\mathcal{T}_{ports} \equiv_{def} (\mathcal{Y}_{fp}\ \ \lambda x_1.\lambda x_2.((\mathcal{F}_{zero}\ x_2)\ 0\ (p_{\langle x_2 \rangle} \oplus (x_1\ (\mathcal{F}_{pred}\ x_3)))))\ )$$

Here $p_{\langle \mathcal{C}_i \rangle}$ reduces to p_i for all integers i by a new reduction rule that applies to the extended calculus. This rule is written $p_{\langle \mathcal{C}_i \rangle} \twoheadrightarrow_\delta p_i$. The same rule was applied in the parallel composition example as well. It also applies to attributes and relations, and it may be interleaved they may be interleaved with β reductions. As an example

$$l_{\langle \mathcal{C}_3 \rangle}\{p_2 := r_{\langle \mathcal{F}_{pred}\ \mathcal{C}_1 \rangle}\} \twoheadrightarrow_{\beta\delta} l_3\{p_2 := r_0\}$$

2.2.1 Formal Details of the Extended Calculus

In this section, I provide the formal details of the extended calculus. I define its syntax and extend the notion of structural equivalence over the extended calculus. I demonstrate the increased scalability that higher-order components can provide in Theorem 2.4. I conclude with a proof that the proposition $\mathcal{A} =_{\mathfrak{H}} \mathcal{B}$, which states that terms $\mathcal{A}$ and $\mathcal{B}$ are structurally equivalent in the extended calculus, is not decidable

for all terms in the extended calculus, even though structural equivalence was for the simple calculus. This is the cost of doing business with this more compact syntax and expressive.

Let $\mathfrak{P}$ be the set of all port names, $\mathfrak{R}$ the set of relation names, $\mathfrak{A}$ the set of all attribute names, $\mathfrak{L}$ the set of all component label names, and $\mathfrak{X}$ the set of all variable names. I assume all four sets are mutually disjoint and denumerable and use the naming conventions above, so that $p_0, p_1, \ldots \in \mathfrak{P}$, $r_0, r_1, \ldots \in \mathfrak{R}$, $A_0, A_1, \ldots \in \mathfrak{A}$, $l_0, l_1, \ldots \in \mathfrak{L}$, and $x_0, x_1, \ldots \in \mathfrak{X}$.

I use the same name variable convention above, but note that a name variable, like i, j or k is not the same thing as a variable name like x_2. The first represents an arbitrary integer. The second is a term in the calculus, that may be substituted for another term according to the reduction rules of the calculus. Then x_j represents an arbitrary variable name in the calculus.

I define the set $\mathfrak{H}$ of terms in the extended calculus to be the smallest set such

that

$$
\begin{aligned}
true \Rightarrow \quad & 0 \in \mathfrak{H} \\
x_i \in \mathfrak{X} \Rightarrow \quad & x_i \in \mathfrak{H} \\
p_i \in \mathfrak{P} \Rightarrow \quad & p_i \in \mathfrak{H} \\
r_i \in \mathfrak{R} \Rightarrow \quad & r_i \in \mathfrak{H} \\
A_i \in \mathfrak{A} \Rightarrow \quad & A_i \in \mathfrak{H} \\
l_i \in \mathfrak{L} \ \wedge \ p_j, p_k \in \mathfrak{P} \Rightarrow \quad & l_i\{p_j := p_k\} \in \mathfrak{H} \\
l_i \in \mathfrak{L} \ \wedge \ p_j, \in \mathfrak{P} \ \wedge \ r_k \in \mathfrak{R} \Rightarrow \quad & l_i\{p_j := r_k\} \in \mathfrak{H} \\
\mathcal{T} \in \mathfrak{H} \ \wedge \ \mathcal{S} \in \mathfrak{H} \Rightarrow \quad & (\mathcal{T} \oplus \mathcal{S}) \in \mathfrak{H} \\
\mathcal{T} \in \mathfrak{H} \ \wedge \ l_i \in \mathfrak{L} \Rightarrow \quad & l_i : [\mathcal{T}] \in \mathfrak{H} \\
\mathcal{T} \in \mathfrak{H} \ \wedge \ x_i \in \mathfrak{X} \Rightarrow \quad & (\lambda x_i . \mathcal{T}) \in \mathfrak{H} \\
\mathcal{T} \in \mathfrak{H} \ \wedge \ \mathcal{S} \in \mathfrak{H} \Rightarrow \quad & (\mathcal{T} \ \mathcal{S}) \in \mathfrak{H}
\end{aligned}
$$

and also,

$$
\begin{aligned}
\mathcal{T} \in \mathfrak{H} \Rightarrow \quad & p_{\langle \mathcal{T} \rangle} \in \mathfrak{H} \\
\mathcal{T} \in \mathfrak{H} \Rightarrow \quad & r_{\langle \mathcal{T} \rangle} \in \mathfrak{H} \\
\mathcal{T} \in \mathfrak{H} \Rightarrow \quad & A_{\langle \mathcal{T} \rangle} \in \mathfrak{H} \\
\mathcal{T} \in \mathfrak{H} \ \wedge \ \mathcal{S} \in \mathfrak{H} \Rightarrow \quad & l_{\langle \mathcal{T} \rangle} : [\mathcal{S}] \in \mathfrak{H}
\end{aligned}
$$

Finally, given name variables i, j, k and arbitrary terms $\mathcal{T}, \mathcal{S}, \mathcal{U} \in \mathfrak{H}$, let

$$\epsilon \equiv_{def} \ i \text{ or } \langle \mathcal{T} \rangle \tag{2.5}$$

$$\zeta \equiv_{def} \ j \text{ or } \langle \mathcal{S} \rangle \tag{2.6}$$

$$\gamma \equiv_{def} \ k \text{ or } \langle \mathcal{U} \rangle \tag{2.7}$$

That is to say that ϵ, ζ, γ are either arbitrary integers or arbitrary terms in the calculus, enclosed with angle brackets. Then

$$l_\epsilon\{p_\zeta := p_\gamma\} \in \mathfrak{H}$$

$$l_\epsilon\{p_\zeta := r_\gamma\} \in \mathfrak{H}$$

To be sure, by this I mean that substituting ϵ, ζ, or γ with a natural number or another term surrounded by angle brackets yields in another expression in $\mathfrak{H}$. The purpose of terms in the subscript of a name is to allow the name of a port, attribute, relation, or component label to be parameterizable with another term in the calculus. Note that $\mathfrak{S} \subset \mathfrak{H}$, which means that each term in the simple calculus is a term in the extended calculus.

Parenthesis are a bit trickier in this calculus than in the simple calculus. It will still be okay to omit parenthesis on sum terms $(\mathcal{T} \oplus \mathcal{S})$, since $\oplus$ remains associative and commutative. Just as in λ calculus abstraction associates to the right, so

$$\lambda x_1.\lambda x_2.\cdots\lambda x_n.\mathcal{T} \equiv (\lambda x_1.(\lambda x_2.(\cdots(\lambda x_n.\mathcal{T})\cdots)))$$

On the other hand, application associates to the left, so

$$\mathcal{T}_1\ \mathcal{T}_2\ \cdots\ \mathcal{T}_n \equiv (((\mathcal{T}_1\ \mathcal{T}_2)\ \cdots)\ \mathcal{T}_n)$$

Any operator in the simple calculus binds more tightly than application and abstraction, so for instance

$$\begin{aligned} \lambda x_1.\mathcal{T}_1 \oplus \mathcal{T}_2 &\equiv \lambda x_1.(\mathcal{T}_1 \oplus \mathcal{T}_2) \\ \mathcal{T}_0\ \mathcal{T}_1 \oplus \mathcal{T}_2 &\equiv \mathcal{T}_0\ (\mathcal{T}_1 \oplus \mathcal{T}_2) \end{aligned}$$

Define the *free variables* of a term inductively with:

$$\begin{aligned} \mathrm{FV}(\mathcal{T}) &= \emptyset \quad \text{if} \quad \mathcal{T} \in \mathfrak{S} \\ \mathrm{FV}(x_i) &= \{x_i\} \\ \mathrm{FV}(\mathcal{T} \oplus \mathcal{S}) &= \mathrm{FV}(\mathcal{T}) \cup \mathrm{FV}(\mathcal{S}) \\ \mathrm{FV}(l_i\!:\![\mathcal{T}]) &= \mathrm{FV}(\mathcal{T}) \\ \mathrm{FV}(\lambda x_i.\mathcal{T}) &= \mathrm{FV}(\mathcal{T}) \setminus \{x_i\} \\ \mathrm{FV}(\mathcal{T}\ \mathcal{S}) &= \mathrm{FV}(\mathcal{T}) \cup \mathrm{FV}(\mathcal{S}) \\ \mathrm{FV}(p_{\langle \mathcal{T} \rangle}) &= \mathrm{FV}(\mathcal{T}) \\ \mathrm{FV}(r_{\langle \mathcal{T} \rangle}) &= \mathrm{FV}(\mathcal{T}) \\ \mathrm{FV}(A_{\langle \mathcal{T} \rangle}) &= \mathrm{FV}(\mathcal{T}) \\ \mathrm{FV}(l_{\langle \mathcal{T} \rangle}\!:\![\mathcal{S}]) &= \mathrm{FV}(\mathcal{T}) \cup \mathrm{FV}(\mathcal{S}) \end{aligned}$$

Finally,

$$\mathrm{FV}(l_i\{p_j := p_{\langle\mathcal{T}\rangle}\}) = \mathrm{FV}(T)$$

$$\mathrm{FV}(l_i\{p_{\langle\mathcal{T}\rangle} := p_j\}) = \mathrm{FV}(\mathcal{T})$$

$$\mathrm{FV}(l_i\{p_{\langle\mathcal{T}\rangle} := p_{\langle\mathcal{S}\rangle}\}) = \mathrm{FV}(\mathcal{T}) \cup \mathrm{FV}(\mathcal{S})$$

$$\mathrm{FV}(l_{\langle\mathcal{T}\rangle}\{p_i := p_j\}) = \mathrm{FV}(\mathcal{T})$$

$$\mathrm{FV}(l_{\langle\mathcal{T}\rangle}\{p_i := p_{\langle\mathcal{S}\rangle}\}) = \mathrm{FV}(\mathcal{T}) \cup \mathrm{FV}(\mathcal{S})$$

$$\mathrm{FV}(l_{\langle\mathcal{T}\rangle}\{p_{\langle\mathcal{S}\rangle} := p_i\}) = \mathrm{FV}(\mathcal{T}) \cup \mathrm{FV}(\mathcal{S})$$

$$\mathrm{FV}(l_{\langle\mathcal{T}\rangle}\{p_{\langle\mathcal{S}\rangle} := p_{\langle\mathcal{V}\rangle}\}) = \mathrm{FV}(\mathcal{T}) \cup \mathrm{FV}(\mathcal{S}) \cup \mathrm{FV}(\mathcal{U})$$

and

$$\mathrm{FV}(l_i\{p_j := r_{\langle\mathcal{T}\rangle}\}) = \mathrm{FV}(T)$$

$$\mathrm{FV}(l_i\{p_{\langle\mathcal{T}\rangle} := r_j\}) = \mathrm{FV}(\mathcal{T})$$

$$\mathrm{FV}(l_i\{p_{\langle\mathcal{T}\rangle} := r_{\langle\mathcal{S}\rangle}\}) = \mathrm{FV}(\mathcal{T}) \cup \mathrm{FV}(\mathcal{S})$$

$$\mathrm{FV}(l_{\langle\mathcal{T}\rangle}\{p_i := r_j\}) = \mathrm{FV}(\mathcal{T})$$

$$\mathrm{FV}(l_{\langle\mathcal{T}\rangle}\{p_i := r_{\langle\mathcal{S}\rangle}\}) = \mathrm{FV}(\mathcal{T}) \cup \mathrm{FV}(\mathcal{S})$$

$$\mathrm{FV}(l_{\langle\mathcal{T}\rangle}\{p_{\langle\mathcal{S}\rangle} := r_i\}) = \mathrm{FV}(\mathcal{T}) \cup \mathrm{FV}(\mathcal{S})$$

$$\mathrm{FV}(l_{\langle\mathcal{T}\rangle}\{p_{\langle\mathcal{S}\rangle} := r_{\langle\mathcal{V}\rangle}\}) = \mathrm{FV}(\mathcal{T}) \cup \mathrm{FV}(\mathcal{S}) \cup \mathrm{FV}(\mathcal{U})$$

Given $x_m \in \mathfrak{X}$ and $\mathcal{U} \in \mathfrak{S}$, the expression $\mathcal{T}\lfloor\mathcal{U}/x_m\rfloor$ denotes the expression that

results from simultaneously replacing all free instances of x_m in $\mathcal{T}$ with $\mathcal{U}$. Formally,

$$0\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} 0$$

$$x_i\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} \begin{cases} \mathcal{U} & x_i \equiv x_m \\ x_i & x_i \not\equiv x_m \end{cases}$$

$$p_i\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} p_i \qquad\qquad p_{\langle \mathcal{T} \rangle}\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} p_{\langle \mathcal{T}\lfloor \mathcal{U}/x_m \rfloor \rangle}$$

$$A_i\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} A_i \qquad\qquad A_{\langle \mathcal{T} \rangle}\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} A_{\langle \mathcal{T}\lfloor \mathcal{U}/x_m \rfloor \rangle}$$

$$r_i\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} r_i \qquad\qquad r_{\langle \mathcal{T} \rangle}\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} r_{\langle \mathcal{T}\lfloor \mathcal{U}/x_m \rfloor \rangle}$$

Also

$$l_i\{p_j := p_k\}\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} l_i\{p_j := p_k\}$$

$$l_i\{p_j := p_{\langle \mathcal{T} \rangle}\}\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} l_i\{p_j := p_{\langle \mathcal{T}\lfloor \mathcal{U}/x_m \rfloor \rangle}\}$$

$$l_i\{p_{\langle \mathcal{T} \rangle} := p_j\}\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} l_i\{p_{\langle \mathcal{T}\lfloor \mathcal{U}/x_m \rfloor \rangle} := p_j\}$$

$$l_i\{p_{\langle \mathcal{T} \rangle} := p_{\langle \mathcal{S} \rangle}\}\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} l_i\{p_{\langle \mathcal{T}\lfloor \mathcal{U}/x_m \rfloor \rangle} := p_{\langle \mathcal{S}\lfloor \mathcal{U}/x_m \rfloor \rangle}\}$$

$$l_{\langle \mathcal{T} \rangle}\{p_j := p_k\}\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} l_{\langle \mathcal{T}\lfloor \mathcal{U}/x_m \rfloor \rangle}\{p_j := p_k\}$$

$$l_{\langle \mathcal{T} \rangle}\{p_i := p_{\langle \mathcal{S} \rangle}\}\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} l_{\langle \mathcal{T}\lfloor \mathcal{U}/x_m \rfloor \rangle}\{p_i := p_{\langle \mathcal{S}\lfloor \mathcal{U}/x_m \rfloor \rangle}\}$$

$$l_{\langle \mathcal{T} \rangle}\{p_{\langle \mathcal{S} \rangle} := p_i\}\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} l_{\langle \mathcal{T}\lfloor \mathcal{U}/x_m \rfloor \rangle}\{p_{\langle \mathcal{S}\lfloor \mathcal{U}/x_m \rfloor \rangle} := p_i\}$$

$$l_{\langle \mathcal{T} \rangle}\{p_{\langle \mathcal{S} \rangle} := p_{\langle \mathcal{V} \rangle}\}\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} l_{\langle \mathcal{T}\lfloor \mathcal{U}/x_m \rfloor \rangle}\{p_{\langle \mathcal{S}\lfloor \mathcal{U}/x_m \rfloor \rangle} := p_{\langle \mathcal{V}\lfloor \mathcal{U}/x_m \rfloor \rangle}\}$$

and

$$l_i\{p_j := r_k\}\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} l_i\{p_j := r_k\}$$

$$l_i\{p_j := r_{\langle \mathcal{T} \rangle}\}\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} l_i\{p_j := r_{\langle \mathcal{T}\lfloor \mathcal{U}/x_m \rfloor \rangle}\}$$

$$l_i\{p_{\langle \mathcal{T} \rangle} := r_j\}\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} l_i\{p_{\langle \mathcal{T}\lfloor \mathcal{U}/x_m \rfloor \rangle} := r_j\}$$

$$l_i\{p_{\langle \mathcal{T} \rangle} := r_{\langle \mathcal{S} \rangle}\}\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} l_i\{p_{\langle \mathcal{T}\lfloor \mathcal{U}/x_m \rfloor \rangle} := r_{\langle \mathcal{S}\lfloor \mathcal{U}/x_m \rfloor \rangle}\}$$

$$l_{\langle \mathcal{T} \rangle}\{p_i := r_j\}\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} l_{\langle \mathcal{T}\lfloor \mathcal{U}/x_m \rfloor \rangle}\{p_i := r_j\}$$

$$l_{\langle \mathcal{T} \rangle}\{p_i := r_{\langle \mathcal{S} \rangle}\}\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} l_{\langle \mathcal{T}\lfloor \mathcal{U}/x_m \rfloor \rangle}\{p_i := r_{\langle \mathcal{S}\lfloor \mathcal{U}/x_m \rfloor \rangle}\}$$

$$l_{\langle \mathcal{T} \rangle}\{p_{\langle \mathcal{S} \rangle} := r_i\}\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} l_{\langle \mathcal{T}\lfloor \mathcal{U}/x_m \rfloor \rangle}\{p_{\langle \mathcal{S}\lfloor \mathcal{U}/x_m \rfloor \rangle} := r_i\}$$

$$l_{\langle \mathcal{T} \rangle}\{p_{\langle \mathcal{S} \rangle} := r_{\langle \mathcal{V} \rangle}\}\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} l_{\langle \mathcal{T}\lfloor \mathcal{U}/x_m \rfloor \rangle}\{p_{\langle \mathcal{S}\lfloor \mathcal{U}/x_m \rfloor \rangle} := r_{\langle \mathcal{V}\lfloor \mathcal{U}/x_m \rfloor \rangle}\}$$

Finally,

$$(\mathcal{T} \oplus \mathcal{S})\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} \mathcal{T}\lfloor \mathcal{U}/x_m \rfloor \oplus \mathcal{S}\lfloor \mathcal{U}/x_m \rfloor$$

$$l_i{:}[\mathcal{T}]\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} l_i{:}[\mathcal{T}\lfloor \mathcal{U}/x_m \rfloor]$$

$$l_{\langle \mathcal{S} \rangle}{:}[\mathcal{T}]\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} l_{\langle \mathcal{S}\lfloor \mathcal{U}/x_m \rfloor \rangle}{:}[\mathcal{T}\lfloor \mathcal{U}/x_m \rfloor]$$

$$(\mathcal{T}\ \mathcal{S})\lfloor \mathcal{U}/x_m \rfloor \equiv_{def} (\mathcal{T}\lfloor \mathcal{U}/x_m \rfloor\ \mathcal{S}\lfloor \mathcal{U}/x_m \rfloor)$$

$$(\lambda x_i.\mathcal{T})\lfloor \mathcal{U}/x_m \rfloor \equiv_{def}$$

$$\begin{cases} \lambda x_i.\mathcal{T} & x_i \equiv x_m \\ \lambda x_i.\mathcal{T}\lfloor \mathcal{U}/x_m \rfloor & x_i \not\equiv x_m \ \wedge \ x_i \notin \mathrm{FV}(\mathcal{U}) \\ \lambda x_j.(\mathcal{T}\lfloor x_j/x_i \rfloor)\lfloor \mathcal{U}/x_m \rfloor & x_i \not\equiv x_m \ \wedge \ x_i \in \mathrm{FV}(\mathcal{U}) \ \wedge \\ & j = \min\{j \in \mathbb{N} \mid x_j \notin (\mathrm{FV}(\mathcal{T}) \cup \mathrm{FV}(\mathcal{U}))\} \end{cases}$$

A term $\mathcal{T}$ with $\mathrm{FV}(\mathcal{T}) = \emptyset$ is called a *closed term.*

For any relation $\boldsymbol{R} \subseteq \mathfrak{H} \times \mathfrak{H}$, define the *one-step* $\boldsymbol{R}$*-reduction* $\rightarrow_R$ as the smallest

binary relation on $\mathfrak{H}$ such that

$$
\begin{aligned}
(\mathcal{T},\mathcal{S}) \in \boldsymbol{R} \Rightarrow \quad & \mathcal{T} \rightarrow_R \mathcal{S} \\
\mathcal{T} \rightarrow_R \mathcal{S} \ \wedge\ \mathcal{U} \in \mathfrak{H} \Rightarrow \quad & (\mathcal{U}\ \mathcal{T}) \rightarrow_R (\mathcal{U}\ \mathcal{S}) \\
\mathcal{T} \rightarrow_R \mathcal{S} \ \wedge\ \mathcal{U} \in \mathfrak{H} \Rightarrow \quad & (\mathcal{T}\ \mathcal{U}) \rightarrow_R (\mathcal{S}\ \mathcal{U}) \\
\mathcal{T} \rightarrow_R \mathcal{S} \ \wedge\ x_i \in \mathfrak{X} \Rightarrow \quad & \lambda x_i.\mathcal{T} \rightarrow_R \lambda x_i.\mathcal{S} \\
\mathcal{T} \rightarrow_R \mathcal{S} \ \wedge\ \mathcal{U} \in \mathfrak{H} \Rightarrow \quad & (\mathcal{U} \oplus \mathcal{T}) \rightarrow_R (\mathcal{U} \oplus \mathcal{S}) \\
\mathcal{T} \rightarrow_R \mathcal{S} \ \wedge\ \mathcal{U} \in \mathfrak{H} \Rightarrow \quad & (\mathcal{T} \oplus \mathcal{U}) \rightarrow_R (\mathcal{S} \oplus \mathcal{U}) \\
\mathcal{T} \rightarrow_R \mathcal{S} \ \wedge\ l_i \in \mathfrak{L} \Rightarrow \quad & l_i{:}[\mathcal{T}] \rightarrow_R l_i{:}[\mathcal{T}] \\
\mathcal{T} \rightarrow_R \mathcal{S} \ \wedge\ \mathcal{U} \in \mathfrak{H} \Rightarrow \quad & l_{\langle\mathcal{U}\rangle}{:}[\mathcal{T}] \rightarrow_R l_{\langle\mathcal{U}\rangle}{:}[\mathcal{S}] \\
\mathcal{T} \rightarrow_R \mathcal{S} \ \wedge\ \mathcal{U} \in \mathfrak{H} \Rightarrow \quad & l_{\langle\mathcal{T}\rangle}{:}[\mathcal{U}] \rightarrow_R l_{\langle\mathcal{S}\rangle}{:}[\mathcal{U}] \\
\mathcal{T} \rightarrow_R \mathcal{S} \Rightarrow \quad & p_{\langle\mathcal{T}\rangle} \rightarrow_R p_{\langle\mathcal{S}\rangle} \\
\mathcal{T} \rightarrow_R \mathcal{S} \Rightarrow \quad & r_{\langle\mathcal{T}\rangle} \rightarrow_R r_{\langle\mathcal{S}\rangle} \\
\mathcal{T} \rightarrow_R \mathcal{S} \Rightarrow \quad & A_{\langle\mathcal{T}\rangle} \rightarrow_R A_{\langle\mathcal{S}\rangle}
\end{aligned}
$$

and also, for all combinations of ϵ, ζ, γ defined in Equations 2.5 through 2.7,

$$\mathcal{M} \to_R \mathcal{N} \Rightarrow \quad l_{\langle \mathcal{M} \rangle}\{p_\zeta := p_\gamma\} \to_R l_{\langle \mathcal{N} \rangle}\{p_\zeta := p_\gamma\}$$

$$\mathcal{M} \to_R \mathcal{N} \Rightarrow \quad l_\epsilon\{p_{\langle \mathcal{M} \rangle} := p_\gamma\} \to_R l_\epsilon\{p_{\langle \mathcal{N} \rangle} := p_\gamma\}$$

$$\mathcal{M} \to_R \mathcal{N} \Rightarrow \quad l_\epsilon\{p_\zeta := p_{\langle \mathcal{M} \rangle}\} \to_R l_\epsilon\{p_\zeta := p_{\langle \mathcal{N} \rangle}\}$$

$$\mathcal{M} \to_R \mathcal{N} \Rightarrow \quad l_{\langle \mathcal{M} \rangle}\{p_\zeta := r_\gamma\} \to_R l_{\langle \mathcal{N} \rangle}\{p_\zeta := r_\gamma\}$$

$$\mathcal{M} \to_R \mathcal{N} \Rightarrow \quad l_\epsilon\{p_{\langle \mathcal{M} \rangle} := r_\gamma\} \to_R l_\epsilon\{p_{\langle \mathcal{N} \rangle} := r_\gamma\}$$

$$\mathcal{M} \to_R \mathcal{N} \Rightarrow \quad l_\epsilon\{p_\zeta := r_{\langle \mathcal{M} \rangle}\} \to_R l_\epsilon\{p_\zeta := r_{\langle \mathcal{N} \rangle}\}$$

Define the $\boldsymbol{R}$-*reduction* $\twoheadrightarrow_R$ as the smallest binary relation on $\mathfrak{H}$ such that

$$\mathcal{T} \to_R \mathcal{S} \Rightarrow \quad \mathcal{T} \twoheadrightarrow_R \mathcal{S}$$

$$\mathcal{T} \equiv \mathcal{S} \Rightarrow \quad \mathcal{T} \twoheadrightarrow_R \mathcal{S}$$

$$\mathcal{T} \twoheadrightarrow_R \mathcal{S} \ \wedge \ \mathcal{S} \twoheadrightarrow_R \mathcal{T} \Rightarrow \quad \mathcal{T} \twoheadrightarrow_R \mathcal{S}$$

Define the $\boldsymbol{R}$-*equivalence* $=_R$ as the smallest binary relation on $\mathfrak{H}$ such that

$$\mathcal{T} \twoheadrightarrow_R \mathcal{S} \Rightarrow \quad \mathcal{T} =_R \mathcal{S}$$

$$\mathcal{T} =_R \mathcal{S} \Rightarrow \quad \mathcal{S} =_R \mathcal{T}$$

$$\mathcal{T} =_R \mathcal{S} \ \wedge \ \mathcal{S} =_R \mathcal{U} \Rightarrow \quad \mathcal{T} =_R \mathcal{U}$$

A term $\mathcal{T}$ is an $\boldsymbol{R}$ *redux* if there exists a term $\mathcal{S}$ with $(\mathcal{T}, \mathcal{S}) \in \boldsymbol{R}$. A term $\mathcal{T}$ is an $\boldsymbol{R}$ *normal form* if there does not exist any subterm of $\mathcal{T}$ that is an $\boldsymbol{R}$ redux.

Define $\boldsymbol{\alpha} \subseteq \mathfrak{H} \times \mathfrak{H}$ as

$$\boldsymbol{\alpha} = \left\{ (\lambda x_i.\mathcal{T}, \lambda x_j.\mathcal{S}) \;\middle|\; x_j \notin \mathrm{FV}(\mathcal{T}) \ \wedge \ \mathcal{S} \equiv \mathcal{T}\lfloor x_j / x_i \rfloor \right\}$$

If $\mathcal{T} =_\alpha \mathcal{S}$, then $\mathcal{T}$ is α-*equivalent* to $\mathcal{S}$. Note that if two terms are α-equivalent, a change of bound variables in one will make it syntactically equivalent to the other.

Define the relation $\boldsymbol{\beta} \subseteq \mathfrak{H} \times \mathfrak{H}$ with

$$\boldsymbol{\beta} = \left\{ \left(((\lambda x.\mathcal{T})\ \mathcal{S}),\ \mathcal{T}\lfloor \mathcal{S}/x \rfloor \right) \mid \mathcal{T}, \mathcal{S} \in \mathfrak{H} \right\}$$

For each $i \in \mathbb{N}$ define the Church numeral $\mathcal{C}_i$ by

$$\begin{aligned} \mathcal{C}_0 &\equiv_{def} \lambda x_1.\lambda x_2.x_2 \\ \mathcal{C}_1 &\equiv_{def} \lambda x_1.\lambda x_2.x_1\ x_2 \\ \mathcal{C}_2 &\equiv_{def} \lambda x_1.\lambda x_2.x_1\ (x_1\ x_2) \\ \mathcal{C}_3 &\equiv_{def} \lambda x_1.\lambda x_2.x_1\ (x_1\ (x_1\ x_2)) \\ &\vdots \end{aligned} \tag{2.8}$$

Define the relation $\boldsymbol{\delta} \subseteq \mathfrak{H} \times \mathfrak{H}$ by

$$\begin{aligned} \boldsymbol{\delta} = \ & \left\{ (p_{\langle \mathcal{T} \rangle}, p_i) \middle| \mathcal{T} =_\alpha \mathcal{C}_i \right\} \cup \\ & \left\{ (r_{\langle \mathcal{T} \rangle}, r_i) \middle| \mathcal{T} =_\alpha \mathcal{C}_i \right\} \cup \\ & \left\{ (A_{\langle \mathcal{T} \rangle}, A_i) \middle| \mathcal{T} =_\alpha \mathcal{C}_i \right\} \cup \\ & \left\{ (l_{\langle \mathcal{T} \rangle} : [\mathcal{S}], l_i : [\mathcal{S}]) \middle| \mathcal{T} =_\alpha \mathcal{C}_i \right\} \cup \\ & \left\{ (l_{\langle \mathcal{V} \rangle}\{p_\zeta := r_\gamma\}, l_i\{p_\zeta := r_\gamma\}) \middle| \mathcal{V} =_\alpha \mathcal{C}_i \right\} \cup \\ & \left\{ (l_\epsilon\{p_{\langle \mathcal{V} \rangle} := r_\gamma\}, l_\epsilon\{p_j := r_\gamma\}) \middle| \mathcal{V} =_\alpha \mathcal{C}_j \right\} \cup \\ & \left\{ (l_\epsilon\{p_\zeta := r_{\langle \mathcal{V} \rangle}\}, l_\epsilon\{p_\zeta := r_k\}) \middle| \mathcal{V} =_\alpha \mathcal{C}_k \right\} \end{aligned}$$

where ϵ, ζ, and γ are again any possible value in Equations 2.5 through 2.7.

Define $\boldsymbol{\beta\delta} \subseteq \mathcal{H} \times \mathcal{H}$ as

$$\boldsymbol{\beta\delta} = \boldsymbol{\alpha} \cup \boldsymbol{\beta} \cup \boldsymbol{\delta}$$

I call this $\boldsymbol{\beta\delta}$ to match up with the equivalent definition in λ calculus. The reason for omitting $\boldsymbol{\alpha}$ is that α reductions are typically only used to identify two terms after a series of β and δ reductions. For instance, if

$$\mathcal{T} \twoheadrightarrow_{\beta\delta} \lambda x_1.x_1$$

$$\mathcal{S} \twoheadrightarrow_{\beta\delta} \lambda x_2.x_2$$

it is only true that $\mathcal{T} =_{\beta\delta} \mathcal{S}$ when the $\boldsymbol{\alpha}$ rule is included, since $\lambda x_1.x_1 \rightarrow_\alpha \lambda x_2.x_2$.

2.2.2 Reduction Isomorphism to λ Calculus with Constants

In this section I prove that the extended calculus is isomorphic to an extension of λ calculus with a set of constants and δ reduction rules. I will prove this and prove that this extension satisfies the Church Rosser property, which, by isomorphism, will prove that $\twoheadrightarrow_{\beta\delta}$ satisfies the Church Rosser property. A binary relation $\boldsymbol{R}$ satisfies the Church Rosser property if

$$\forall M, M_1, M_2.\ M\boldsymbol{R}M_1 \ \wedge\ M\boldsymbol{R}M_2 \Rightarrow \quad \exists M_3.\ M_1\boldsymbol{R}M_3 \ \wedge\ M_2\boldsymbol{R}M_3$$

This property, applied to the extended calculus, says that if $\mathcal{S} \in \mathfrak{H}$, and $\beta\delta$ reduces to term $\mathcal{S}_1$ and term $\mathcal{S}_2$, then there must be some term $\mathcal{T} \in \mathfrak{H}$ such that both $\mathcal{S}_1$ and $\mathcal{S}_2$ $\beta\delta$ reduce to $\mathcal{T}$. This is important, because there may be many different ways to reduce the same expression in this calculus. One immediate consequence of Church

Rosser is that each term in the extended calculus to reduces to at most one term in the simple calculus, or each term can represent at most one structure. This is because each term in the simple calculus is irreducible.

Given a set $\mathfrak{C}$ of constants, and a set $\mathfrak{X}$ of variables, the set of λ terms $\Lambda(\mathfrak{C})$ is the smallest set such that

$$\begin{aligned} x_i \in \mathfrak{X} \Rightarrow \quad & x_i \in \Lambda(\mathfrak{C}) \\ c_i \in \mathfrak{C} \Rightarrow \quad & c_i \in \Lambda(\mathfrak{C}) \\ \mathcal{S}, \mathcal{T} \in \Lambda(\mathfrak{C}) \Rightarrow \quad & (\mathcal{S}\ \mathcal{T}) \in \Lambda(\mathfrak{C}) \\ x_i \in \mathfrak{X}\ \wedge\ \mathcal{S} \in \Lambda(\mathfrak{C}) \Rightarrow \quad & (\lambda x_i.\mathcal{S}) \in \Lambda(\mathfrak{C}) \end{aligned}$$

This extension is nearly identical to standard untyped λ calculus, but now there is a special constant $\delta \in \mathfrak{C}$. For some finite set of n-ary relations $R_1, \ldots, R_m \subseteq \Lambda(\mathfrak{C})^n$ and corresponding functions $g_1, \ldots, g_m$ with $g_i \in R_i \to \Lambda(\mathfrak{C})$, you may define a binary relation $\boldsymbol{\delta}$ on $\Lambda(\mathfrak{C})$ with

$$\begin{aligned} \boldsymbol{\delta} &= \big\{((\delta\ \mathcal{T}_1\ \cdots \mathcal{T}_n), g_1(\mathcal{T}_1, \ldots, \mathcal{T}_n)) \big| (\mathcal{T}_1, \ldots, \mathcal{T}_n) \in R_1\big\} \\ &\vdots \\ \boldsymbol{\delta} &= \big\{((\delta\ \mathcal{T}_1\ \cdots \mathcal{T}_n), g_m(\mathcal{T}_1, \ldots, \mathcal{T}_n)) \big| (\mathcal{T}_1, \ldots, \mathcal{T}_n) \in R_m\big\} \end{aligned}$$

Barendregt [5] provides a full description of this calculus, but this rule $\boldsymbol{\delta}$ is what makes the extension interesting. Adding such a rule may cause the corresponding reduction rule $\twoheadrightarrow_{\beta\delta}$ to violate the Church Rosser property. As an example, let $\mathfrak{C} = \{\delta, \epsilon\}$ and

$$\boldsymbol{\delta} = \big\{((\delta\ \mathcal{T}\ \mathcal{T}), \epsilon) \big| \mathcal{T} \in \Lambda(\mathfrak{C})\big\}$$

Theorem 2.3. *There exists a set of constants* $\mathfrak{C}$*, a set of* δ *reductions over* $\Lambda(\mathfrak{C})$*, and a bijective map* $f : \mathfrak{H} \to Y$*, where* Y *is a subset of* $\Lambda(\mathfrak{C})$ *closed under* $\twoheadrightarrow_{\beta\delta}$*, such that for each* $\mathcal{S} \in \mathfrak{H}$*,* $\mathcal{S} \twoheadrightarrow_{\beta\delta} \mathcal{T}$ *if and only if* $f(\mathcal{S}) \twoheadrightarrow_{\beta\delta} f(\mathcal{T})$*. Moreover, the relation* $\twoheadrightarrow_{\beta\delta} \subseteq \Lambda(\mathfrak{C}) \times \Lambda(\mathfrak{C})$ *satisfies the Church Rosser property.*

Proof. Let

$$\mathfrak{C} = \mathfrak{P} \cup \mathfrak{R} \cup \mathfrak{A} \cup \mathfrak{L} \cup \{0,\ c_{:=}\ ,\ c_{\oplus}\ ,\ c_{:[\ ]}\ ,\ c_p\ ,\ c_r\ ,\ c_A\ ,\ c_l, \delta\}$$

Define

$$\begin{aligned} \boldsymbol{\delta} &= \left\{((\delta\ c_p\ \mathcal{U}), p_i) \middle| \mathcal{U} \equiv \mathcal{C}_i\right\} \\ &= \left\{((\delta\ c_r\ \mathcal{U}), r_i) \middle| \mathcal{U} \equiv \mathcal{C}_i\right\} \\ &= \left\{((\delta\ c_A\ \mathcal{U}), A_i) \middle| \mathcal{U} \equiv \mathcal{C}_i\right\} \\ &= \left\{((\delta\ c_l\ \mathcal{U}), l_i) \middle| \mathcal{U} \equiv \mathcal{C}_i\right\} \end{aligned}$$

Define f on $\mathfrak{H}$ inductively as follows:

$$f(0) \equiv_{def} 0$$

$$f(x_i) \equiv_{def} x_i$$

$$f(p_i) \equiv_{def} p_i \qquad f(p_{\langle \mathcal{T} \rangle}) \equiv_{def} (\delta\ c_p\ \mathcal{T})$$

$$f(r_i) \equiv_{def} r_i \qquad f(r_{\langle \mathcal{T} \rangle}) \equiv_{def} (\delta\ c_r\ \mathcal{T})$$

$$f(A_i) \equiv_{def} A_i \qquad f(A_{\langle \mathcal{T} \rangle}) \equiv_{def} (\delta\ c_A\ \mathcal{T})$$

Also

$$f(l_i\{p_j := p_k\}) \equiv_{def} (c_{:=}\ \ l_i\ p_j\ p_k)$$

$$f(l_i\{p_j := p_{\langle\mathcal{T}\rangle}\}) \equiv_{def} (c_{:=}\ \ l_i\ p_j\ (\delta\ c_p\ \mathcal{T}))$$

$$f(l_i\{p_{\langle\mathcal{T}\rangle} := p_j\}) \equiv_{def} (c_{:=}\ \ l_i\ (\delta\ c_p\ \mathcal{T})\ p_j)$$

$$f(l_i\{p_{\langle\mathcal{T}\rangle} := p_{\langle\mathcal{S}\rangle}\}) \equiv_{def} (c_{:=}\ \ l_i\ (\delta\ c_p\ \mathcal{T})\ (\delta\ c_p\ \mathcal{S}))$$

$$f(l_{\langle\mathcal{T}\rangle}\{p_i := p_j\}) \equiv_{def} (c_{:=}\ \ (\delta\ c_l\ \mathcal{T})\ p_i\ p_j)$$

$$f(l_{\langle\mathcal{T}\rangle}\{p_i := p_{\langle\mathcal{S}\rangle}\}) \equiv_{def} (c_{:=}\ \ (\delta\ c_l\ \mathcal{T})\ p_i\ (\delta\ c_p\ \mathcal{S}))$$

$$f(l_{\langle\mathcal{T}\rangle}\{p_{\langle\mathcal{S}\rangle} := p_i\}) \equiv_{def} (c_{:=}\ \ (\delta\ c_l\ \mathcal{T})\ (\delta\ c_p\ \mathcal{S})\ p_i)$$

$$f(l_{\langle\mathcal{T}\rangle}\{p_{\langle\mathcal{S}\rangle} := p_{\langle\mathcal{V}\rangle}\}) \equiv_{def} (c_{:=}\ \ (\delta\ c_l\ \mathcal{T})\ (\delta\ c_p\ \mathcal{S})\ (\delta\ c_p\ \mathcal{V}))$$

and

$$f(l_i\{p_j := r_k\}) \equiv_{def} (c_{:=}\ \ l_i\ p_j\ r_k)$$

$$f(l_i\{p_j := r_{\langle\mathcal{T}\rangle}\}) \equiv_{def} (c_{:=}\ \ l_i\ p_j\ (\delta\ c_r\ \mathcal{T}))$$

$$f(l_i\{p_{\langle\mathcal{T}\rangle} := r_j\}) \equiv_{def} (c_{:=}\ \ l_i\ (\delta\ c_p\ \mathcal{T})\ r_j)$$

$$f(l_i\{p_{\langle\mathcal{T}\rangle} := r_{\langle\mathcal{S}\rangle}\}) \equiv_{def} (c_{:=}\ \ l_i\ (\delta\ c_p\ \mathcal{T})\ (\delta\ c_r\ \mathcal{S}))$$

$$f(l_{\langle\mathcal{T}\rangle}\{p_i := r_j\}) \equiv_{def} (c_{:=}\ \ (\delta\ c_l\ \mathcal{T})\ p_i\ r_j)$$

$$f(l_{\langle\mathcal{T}\rangle}\{p_i := r_{\langle\mathcal{S}\rangle}\}) \equiv_{def} (c_{:=}\ \ (\delta\ c_l\ \mathcal{T})\ p_i\ (\delta\ c_r\ \mathcal{S}))$$

$$f(l_{\langle\mathcal{T}\rangle}\{p_{\langle\mathcal{S}\rangle} := r_i\}) \equiv_{def} (c_{:=}\ \ (\delta\ c_l\ \mathcal{T})\ (\delta\ c_p\ \mathcal{S})\ r_i)$$

$$f(l_{\langle\mathcal{T}\rangle}\{p_{\langle\mathcal{S}\rangle} := r_{\langle\mathcal{V}\rangle}\}) \equiv_{def} (c_{:=}\ \ (\delta\ c_l\ \mathcal{T})\ (\delta\ c_p\ \mathcal{S})\ (\delta\ c_r\ \mathcal{V}))$$

Finally,

$$f(\mathcal{T} \oplus \mathcal{S}) \equiv_{def} (c_{\oplus}\ \mathcal{T}\ \mathcal{S})$$

$$f(l_i{:}[\mathcal{T}]) \equiv_{def} (c_{:[\ ]}\ \ l_i\ \mathcal{T})$$

$$f(l_{\langle\mathcal{S}\rangle}{:}[\mathcal{T}]) \equiv_{def} (c_{:[\ ]}\ \ (\delta\ c_l\ \mathcal{S})\ \mathcal{T})$$

$$f(\lambda x_i.\mathcal{T}) \equiv_{def} (\lambda x_i.\mathcal{T})$$

$$f(\mathcal{T}\ \mathcal{S}) \equiv_{def} (\mathcal{T}\ \mathcal{S})$$

It is easy to see that f is injective. Let $Y = \mathrm{rng}(f)$. Clearly $Y \subseteq \Lambda(\mathfrak{C})$. Given $\mathcal{S}, \mathcal{T} \in \mathfrak{S}$, note that $\mathcal{S} \rightarrow_\beta \mathcal{T}$ if and only if $f(\mathcal{S}) \rightarrow_\beta f(\mathcal{T})$, since β reduction is no different in $\Lambda(\mathfrak{C})$. Also, by our parallel definitions of $\rightarrow_\delta$ over $\mathfrak{H}$ and $\Lambda(\mathfrak{C})$, $\mathcal{S} \rightarrow_\delta \mathcal{T}$ if and only if $f(\mathcal{S}) \rightarrow_\delta f(\mathcal{T})$. Note that Y must then be closed under $\rightarrow_\beta$ and $\rightarrow_\delta$.

By transitivity, Y is closed under $\twoheadrightarrow_{\beta\delta}$. Of course, f, viewed as a function from $\mathfrak{H}$ to Y, must be bijective, since it is injective and Y is its range. Suppose that $\mathcal{U}, \mathcal{V} \in Y$, and $\mathcal{U} \twoheadrightarrow_{\beta\delta} \mathcal{V}$. Then there must be a $\mathcal{U}_0, \mathcal{U}_1, \ldots, \mathcal{U}_n \in Y$, with $\mathcal{U}_0 \equiv \mathcal{U}$ and $\mathcal{U}_n \equiv \mathcal{T}$, such that

$$\mathcal{U}_0 \rightarrow_{\gamma_0} \mathcal{U}_1 \rightarrow_{\gamma_{n-1}} \cdots \rightarrow_{\gamma} \mathcal{U}_n$$

where each γ_i is either β or δ. Clearly $f^{-1}(\mathcal{U}_0) \twoheadrightarrow_{\beta\delta} f^{-1}(\mathcal{U}_0)$. Suppose $f^{-1}(\mathcal{U}_0) \twoheadrightarrow_{\beta\delta} f^{-1}(\mathcal{U}_m)$ for $m < n$. If γ_m is β, than $f^{-1}(\mathcal{U}_m) \rightarrow_\beta f^{-1}(\mathcal{U}_{m+1})$. If γ_m is δ, than $f^{-1}(\mathcal{U}_m) \rightarrow_\delta f^{-1}(\mathcal{U}_{m+1})$. This implies $f^{-1}(\mathcal{U}_0) \rightarrow f^{-1}(\mathcal{U}_{m+1})$. Thus, $f^{-1}(\mathcal{U}) \twoheadrightarrow_{\beta\delta} f^{-1}(\mathcal{V})$.

The Church Rosser property follows immediately from Theorem 6.3 of [6] since $\boldsymbol{\delta}$

is in fact a function on closed normal forms. □

2.2.3 Scalability of Higher-Order Composition

To show the scalability benefit of the extended calculus, there needs to be a definition of the *size* of a term in $\mathcal{H}$. Let $s : \mathcal{H} \to \{1, 2, \ldots\}$ be defined inductively by

$$
\begin{aligned}
s(0) &= 1 \\
s(x_i) &= 1 \\
s(p_i) &= 1 & s(p_{\langle \mathcal{T} \rangle}) &= 2 + s(\mathcal{T}) \\
s(r_i) &= 1 & s(r_{\langle \mathcal{T} \rangle}) &= 2 + s(\mathcal{T}) \\
s(A_i) &= 1 & s(A_{\langle \mathcal{T} \rangle}) &= 2 + s(\mathcal{T})
\end{aligned}
$$

Also

$$s(l_i\{p_j := p_k\}) = 4$$

$$s(l_i\{p_j := p_{\langle\mathcal{T}\rangle}\}) = 6 + s(\mathcal{T})$$

$$s(l_i\{p_{\langle\mathcal{T}\rangle} := p_j\}) = 6 + s(\mathcal{T})$$

$$s(l_i\{p_{\langle\mathcal{T}\rangle} := p_{\langle\mathcal{S}\rangle}\}) = 8 + s(\mathcal{T}) + s(\mathcal{S})$$

$$s(l_{\langle\mathcal{T}\rangle}\{p_i := p_j\}) = 6 + s(\mathcal{T})$$

$$s(l_{\langle\mathcal{T}\rangle}\{p_i := p_{\langle\mathcal{S}\rangle}\}) = 8 + s(\mathcal{T}) + s(\mathcal{S})$$

$$s(l_{\langle\mathcal{T}\rangle}\{p_{\langle\mathcal{S}\rangle} := p_i\}) = 8 + s(\mathcal{T}) + s(\mathcal{S})$$

$$s(l_{\langle\mathcal{T}\rangle}\{p_{\langle\mathcal{S}\rangle} := p_{\langle\mathcal{V}\rangle}\}) = 10 + s(\mathcal{T}) + s(\mathcal{S}) + s(\mathcal{V})$$

and

$$s(l_i\{p_j := r_k\}) = 4$$

$$s(l_i\{p_j := r_{\langle\mathcal{T}\rangle}\}) = 6 + s(\mathcal{T})$$

$$s(l_i\{p_{\langle\mathcal{T}\rangle} := r_j\}) = 6 + s(\mathcal{T})$$

$$s(l_i\{p_{\langle\mathcal{T}\rangle} := r_{\langle\mathcal{S}\rangle}\}) = 8 + s(\mathcal{T}) + s(\mathcal{S})$$

$$s(l_{\langle\mathcal{T}\rangle}\{p_i := r_j\}) = 6 + s(\mathcal{T})$$

$$s(l_{\langle\mathcal{T}\rangle}\{p_i := r_{\langle\mathcal{S}\rangle}\}) = 8 + s(\mathcal{T}) + s(\mathcal{S})$$

$$s(l_{\langle\mathcal{T}\rangle}\{p_{\langle\mathcal{S}\rangle} := r_i\}) = 8 + s(\mathcal{T}) + s(\mathcal{S})$$

$$s(l_{\langle\mathcal{T}\rangle}\{p_{\langle\mathcal{S}\rangle} := r_{\langle\mathcal{V}\rangle}\}) = 10 + s(\mathcal{T}) + s(\mathcal{S}) + s(\mathcal{V})$$

Finally,

$$s(\mathcal{T} \oplus \mathcal{S}) = 1 + s(\mathcal{T}) + s(\mathcal{S})$$

$$s(l_i : [\mathcal{T}]) = 2 + s(\mathcal{T})$$

$$s(l_{\langle \mathcal{S} \rangle} : [\mathcal{T}]) = 3 + s(\mathcal{S}) + s(\mathcal{T})$$

$$s(\lambda x_i.\mathcal{T}) = 1 + s(\mathcal{T})$$

$$s(\mathcal{T}\ \mathcal{S}) = s(\mathcal{T}) + s(\mathcal{S})$$

This definition of size corresponds to the standard definition of size for the equivalent terms in the λ calculus extension of the previous section.

A term $\mathcal{S} \in \mathcal{H}$ is said to be *minimal with respect to* $\mathfrak{H}$ if for all $\mathcal{T} \in \mathfrak{H}$, $s(\mathcal{S}) \leq s(\mathcal{T})$. If $\mathcal{S}$ also belongs to $\mathfrak{S}$, it is said to be *minimal with respect to* $\mathfrak{S}$ if for all $\mathcal{T} \in \mathfrak{S}$, $s(\mathcal{S}) \leq s(\mathcal{T})$.

The main theorem of this section concerns the growth of a class of functions. As in [16], given a function $f : \mathbb{N} \to \mathbb{N}$, I define

$$\Theta(f) = \big\{ g : \mathbb{N} \to \mathbb{N} \big| \exists C, D \in \mathbb{R}_+, n, m \in \mathbb{N}.(\forall k > n.g(k) \geq Cf(k)) \\ \wedge\ (\forall k > m.g(k) \leq Df(k)) \big\}$$

Note here that $\mathbb{N}$ is the set of natural numbers and $\mathbb{R}_+$ is the set of positive real numbers. Similarly,

$$\Omega(f) = \big\{ g : \mathbb{N} \to \mathbb{N} \big| \exists C \in \mathbb{R}_+, n \in \mathbb{N}.\forall k > n.g(k) \geq Cf(k) \big\}$$

and

$$O(f) = \left\{ g : \mathbb{N} \to \mathbb{N} \middle| \exists C \in \mathbb{R}_+, n \in \mathbb{N}. \forall k > n. g(k) \leq Cf(k) \right\}$$

The growth of functions I consider may be polynomial or they may exponential, double exponential, triple exponential, and so on. Given $k \in \{1, 2, \ldots\}$, let $p_k : \mathbb{N} \to \mathbb{N}$ be defined for all $n \in \mathbb{N}$ as

$$p_k(n) = n^k$$

Any function $f \in \Theta(p_k)$ is said to *grow polynomially with order* k. If $k = 1$, it grows *linearly*. Define $\alpha : \{1, 2, \ldots\} \times \mathbb{N} \to \mathbb{N}$ for all $k \in \{1, 2, \ldots\}, n \in \mathbb{N}$ as

$$\alpha(k, n) = \begin{cases} 2^n & k = 1 \\ 2^{\alpha(k-1,n)} & k > 1 \end{cases}$$

For each $k \in \{1, 2, \ldots\}$, let $e_k : \mathbb{N} \to \mathbb{N}$ be defined for all $n \in \mathbb{N}$ as

$$e_k(n) = \alpha(k, n)$$

Note that $e_1(n) = 2^n$, $e_2(n) = 2^{2^n}$, and so on. If $f \in \Theta(e_k)$, it is said to grow *k-tuple exponentially*. If $k = 1$, it is said to grow *exponentially*.

Theorem 2.4. *For any $k \in \mathbb{N}$, there exists a denumerable set of terms $\{\mathcal{S}_0, \mathcal{S}_1, \ldots\}$ in $\mathfrak{S}$, each of which is minimal with respect to $\mathfrak{S}$, and a denumerable set of terms $\{\mathcal{T}_0, \mathcal{T}_1, \ldots\}$ in $\mathfrak{H}$, with each $\mathcal{T}_i =_{\mathfrak{H}} \mathcal{S}_i$, such that $f_{\mathfrak{H}} : \mathbb{N} \to \mathbb{N}$ defined for all $n \in \mathbb{N}$ as*

$$f_{\mathfrak{H}}(n) = s(\mathcal{T}_n)$$

belongs to $O(p_1)$ *whereas* $f_{\mathfrak{S}} : \mathbb{N} \to \mathbb{N}$ *defined for all* $n \in \mathbb{N}$ *as*

$$f_{\mathfrak{S}}(n) = s(\mathcal{S}_n)$$

belongs to $\Omega(e_k)$*. That is to say, the size of the minimal terms* $\mathcal{S}_i$ *in the simple calculus grows at least* k*-tuple exponentially with* i*, whereas the size of their counterparts* $\mathcal{T}_i$ *in the extended calculus grows at most linearly with* i*.*

Proof. Consider the $k = 1$ case first. Let

$$\mathcal{S}_n^1 \equiv_{def} \begin{cases} 0 & n = 0 \\ l_1 : [\mathcal{S}_{n-1}^1] \oplus l_2 : [\mathcal{S}_{n-1}^1] & n > 0 \end{cases}$$

Then

$$\begin{aligned} \mathcal{S}_0^1 &\equiv 0 \\ \mathcal{S}_1^1 &\equiv l_1 : [0] \oplus l_2[0] \\ \mathcal{S}_2^1 &\equiv l_1 : \big[l_1 : [0] \oplus l_2[0]\big] \oplus l_2 : \big[l_1 : [0] \oplus l_2[0]\big] \\ &\vdots \end{aligned}$$

The size of the terms in this case is

$$s(\mathcal{S}_n^1) = \begin{cases} 1 & n = 0 \\ 5 + 2s(\mathcal{S}_{n-1}^1) & n > 0 \end{cases}$$

It is easy to see that for each n, $s(\mathcal{S}_n^1) \geq 2^n$. Let $f_{\mathfrak{S}}^1 : \mathbb{N} \to \mathbb{N}$ for each n as $f_{\mathfrak{S}}^1(n) = s(\mathcal{S}_n^1)$. Then $f_{\mathfrak{S}}^1 \in \Omega(e_1)$.

It is easy to see that$\mathcal{S}_0^1$ is minimal with respect to $\mathfrak{S}$. If $\mathcal{S}_n^1$ is minimal with respect to $\mathfrak{S}$, then

$$\mathcal{S}_{n+1}^1 \equiv l_1 : [\mathcal{S}_n^1] \oplus l_2 : [\mathcal{S}_n^1]$$

must also be minimal with respect to $\mathfrak{S}$. By induction, each $\mathcal{S}_n^1$ is minimal with respect to $\mathfrak{S}$ for all n.

Define $\mathcal{U}$ by

$$\mathcal{U} \equiv_{def} \mathcal{Y}_{fp}\ \lambda x_1 \lambda x_2 . (\mathcal{F}_{zero}\ x_2)\ 0\ (l_1 : [(x_1\ (\mathcal{F}_{pred}\ x_2))] \oplus l_2 : [(x_1\ (\mathcal{F}_{pred}\ x_2))])$$

This is a term I will use here and for the $k > 1$ case. Referring to Equations 2.2, 2.3, and 2.4,

$$s(\mathcal{U}) = 43$$

Now define $\mathcal{T}_n^1$ by

$$\mathcal{T}_n^1 \equiv_{def} (\mathcal{U}\ \mathcal{C}_n)$$

Since $s(\mathcal{C}_n) = 3 + n$, for each n, $s(\mathcal{T}_n^1) = 46 + n$. If I define $f_{\mathfrak{H}}^1$ for all n as

$$f_{\mathfrak{H}}^1 = s(\mathcal{T}_n^1)$$

then $f_{\mathfrak{H}}^1 \in O(p_1)$.

Note that

$$\begin{aligned}
\mathcal{T}_0^1 \quad &\twoheadrightarrow_{\beta\delta} \quad (\lambda x_1 \lambda x_2.(\mathcal{F}_{zero}\ x_2)\ 0\ l_1\!:\![(x_1\ (\mathcal{F}_{pred}\ x_2))] \oplus l_2\!:\![(x_1\ (\mathcal{F}_{pred}\ x_2))]) \\
&\qquad\quad (\mathcal{Y}_{fp}\ \lambda x_1 \lambda x_2.(\mathcal{F}_{zero}\ x_2)\ 0\ l_1\!:\![(x_1\ (\mathcal{F}_{pred}\ x_2))] \oplus l_2\!:\![(x_1\ (\mathcal{F}_{pred}\ x_2))]) \\
&\qquad\quad \mathcal{C}_0 \\
&\twoheadrightarrow_{\beta\delta} \quad 0 \\
&\equiv \quad \mathcal{S}_0^1
\end{aligned}$$

Now suppose $\mathcal{T}_n^1 \rightarrow \mathcal{S}_n^1$. Then

$$\begin{aligned}
\mathcal{T}_{n+1}^1 \quad &\twoheadrightarrow_{\beta\delta} \quad (\lambda x_1 \lambda x_2.(\mathcal{F}_{zero}\ x_2)\ 0\ l_1\!:\![(x_1\ (\mathcal{F}_{pred}\ x_2))] \oplus l_2\!:\![(x_1\ (\mathcal{F}_{pred}\ x_2))]) \\
&\qquad\quad (\mathcal{Y}_{fp}\ \lambda x_1 \lambda x_2.(\mathcal{F}_{zero}\ x_2)\ 0\ l_1\!:\![(x_1\ (\mathcal{F}_{pred}\ x_2))] \oplus l_2\!:\![(x_1\ (\mathcal{F}_{pred}\ x_2))]) \\
&\qquad\quad \mathcal{C}_{n+1} \\
&\twoheadrightarrow_{\beta\delta} \quad l_1\!:\![(\mathcal{U}\ \mathcal{C}_n)] \oplus l_2\!:\![(\mathcal{U}\ \mathcal{C}_n)] \\
&\equiv \quad l_1\!:\![\mathcal{T}_n^1] \oplus l_2\!:\![\mathcal{T}_n^1] \\
&\twoheadrightarrow_{\beta\delta} \quad l_1\!:\![\mathcal{S}_n^1] \oplus l_2\!:\![\mathcal{S}_n^1] \\
&\equiv \quad \mathcal{S}_{n+1}^1
\end{aligned}$$

Thus, for each n, $\mathcal{T}_n^1 \rightarrow \mathcal{S}_n^1$. Since each $\mathcal{S}_n^1$ is in $\boldsymbol{\beta\delta}$ normal form, I can conclude $\mathcal{T}_n^1 =_{\mathfrak{H}} \mathcal{S}_n^1$ (by the Church Rosser property).

When $k > 1$, let

$$\mathcal{S}_n^k \equiv_{def} \mathcal{S}_{\alpha(k,n)}^1 \tag{2.9}$$

Define $f^k_{\mathfrak{S}} : \mathbb{N} \to \mathbb{N}$ for each n as

$$f^k_{\mathfrak{S}}(n) = s(\mathcal{S}^k_n)$$

Then

$$f^k_{\mathfrak{S}}(n) = f^1_{\mathfrak{S}}(\alpha(k, n))$$

Since $f^1_{\mathfrak{S}} \in \Omega(e_1)$, $f^k_{\mathfrak{S}} \in \Omega(e_k)$.

Before defining $\mathcal{T}^k_n$, note that

$$\mathcal{F}_{exp} \equiv_{def} \lambda x_1.\lambda x_2.x_2\ x_1$$

is the exponent function. It is straightforward to verify that for all $m, n \in \mathbb{N}$,

$$\mathcal{F}_{exp}\ \mathcal{C}_m\ \mathcal{C}_n \to \mathcal{C}_{m^n}$$

From this define

$$\mathcal{F}_\alpha \equiv_{def} \mathcal{Y}_{fp}\ \lambda x_1.\lambda x_2.(\mathcal{F}_{zero}\ x_2)\ (\mathcal{F}_{exp}\ \mathcal{C}_2\ x_2)\ (\mathcal{F}_{exp}\ \mathcal{C}_2\ (x_1\ (\mathcal{F}_{pred}\ x_2)))$$

Then for all $k, n \in \mathbb{N}$,

$$\mathcal{F}_\alpha\ \mathcal{C}_k\ \mathcal{C}_n \to \mathcal{C}_{\alpha(k+1,n)}$$

Now $s(\mathcal{F}) = 58$.

Now define $\mathcal{T}^k_n$ by

$$\mathcal{T}^k_n \equiv_{def} \mathcal{U}\ (\mathcal{F}_\alpha\ \mathcal{C}_{k-1}\ \mathcal{C}_n)$$

Then

$$\begin{aligned} \mathcal{T}_n^k &\rightarrow \mathcal{U}\ \mathcal{C}_{\alpha(k,n)} \\ &\equiv \mathcal{T}^1_{\alpha(k,n)} \\ &\rightarrow \mathcal{S}^1_{\alpha(k,n)} \\ &\equiv \mathcal{S}_n^k \end{aligned}$$

Define $f^k_{\mathfrak{H}}$ for all n by

$$\begin{aligned} f^k_{\mathfrak{H}}(n) &= s(\mathcal{T}_n^k) \\ &= 106 + k + n \end{aligned}$$

Then for each $k \in \mathbb{N}$, $f^k_{\mathfrak{H}} \in O(p_1)$, yet $f^k_{\mathfrak{H}}$ defined for all n by

$$f^k_{\mathfrak{H}}(n) = s(\mathcal{S}_n^k)$$

belongs to $\Omega(e_k)$ □

2.2.4 Undecidability of Structural Equivalence

The last section showed how a higher-order composition language can lead to a much more compact representation of a structure than one without higher-order components. This does come with a cost. While testing structural equality was decidable in the simple calculus, it becomes undecidable in the extended calculus.

Suppose that given $\mathcal{T} \in \mathfrak{H}$, $\mathcal{T} =_{\beta\delta} \mathcal{S}$ and $\mathcal{T} =_{\beta\delta} \mathcal{U}$. From the Church Rosser theorem, if both $\mathcal{S}$ and $\mathcal{U}$ belong to $\mathfrak{S}$, then $\mathcal{S} \equiv \mathcal{U}$, because terms in the simple

calculus are $\boldsymbol{\beta\delta}$ normal forms. This means that each term in the higher-order calculus is reduction equivalent to at most one term in the simple calculus.

This makes it possible extend the structural equivalence over the extended calculus. I define the *structural equivalence relation* $=_{\mathfrak{H}}$ to be the smallest subset of $\mathfrak{H} \times \mathfrak{H}$ such that

$$\begin{aligned} \mathcal{S} =_{\beta\delta} \mathcal{T} \Rightarrow \quad & \mathcal{S} =_{\mathfrak{H}} \mathcal{T} \\ \mathcal{S} =_{\mathfrak{S}} \mathcal{T} \Rightarrow \quad & \mathcal{S} =_{\mathfrak{H}} \mathcal{T} \\ \mathcal{S} =_{\mathfrak{H}} \mathcal{T} \ \wedge \ \mathcal{T} =_{\mathfrak{H}} \mathcal{U} \Rightarrow \quad & \mathcal{S} =_{\mathfrak{H}} \mathcal{U} \end{aligned}$$

It is easy to check that $=_{\mathfrak{H}}$ is an equivalence relation and it is closed under $\twoheadrightarrow_{\beta\delta}$. That is if $\mathcal{S} \twoheadrightarrow_{\beta\delta} \mathcal{T}$, then $\mathcal{S} =_{\mathfrak{H}} \mathcal{T}$.

Theorem 2.5. *The proposition $\mathcal{S} =_{\mathfrak{H}} \mathcal{T}$ is undecidable for arbitrary terms in $\mathfrak{H}$.*

Proof. Since $\twoheadrightarrow_{\beta\delta} \subseteq \Lambda(\mathfrak{C}) \times \Lambda(\mathfrak{C})$ is Church Rosser, the Scott-Curry theorem [25] extends to $\twoheadrightarrow_{\beta\delta}$ by Theorem 6.6.6 of [5]. This theorem states that no pair of disjoint sets in $\Lambda(\mathfrak{C})$ closed under $\twoheadrightarrow_{\beta\delta}$ is recursively separable. Consider the mapping f and subset Y of $\Lambda(\mathfrak{C})$ in Theorem 2.3. Given $\mathcal{T} \in \mathfrak{H}$, let

$$Y_t = \{\mathcal{U} \in Y | f^{-1}(\mathcal{U}) =_{\mathfrak{H}} \mathcal{T}\}$$

Note that for all $\mathcal{S} \in \mathfrak{H}$, $\mathcal{S} =_{\mathfrak{H}} \mathcal{T}$ if and only if $f(\mathcal{S}) \in Y_t$, and $\mathcal{S} \in \mathfrak{H}$, $\mathcal{S} \neq_{\mathfrak{H}} \mathcal{T}$ if and only if $f(\mathcal{S}) \in Y \setminus Y_t$. From Theorem 2.3 and the closure property of $=_{\mathfrak{H}}$, Y_t and $Y \setminus Y_t$ are disjoint sets closed under $\rightarrow_{\beta\delta}$. From the Scott-Curry theorem they

are not recursively separable. Hence there is no decision procedure which will always determine whether $f(\mathcal{S})$ belongs to Y_t or $Y \setminus Y_t$ or equivalently whether $\mathcal{S} =_{\mathfrak{H}} \mathcal{T}$ or $\mathcal{S} \neq_{\mathfrak{H}} \mathcal{T}$. □

2.2.5 Abstract Syntax for the Extended Calculus

As a practical matter, higher-order components are used to specify non-higher-order systems. This is reflected in the extended calculus by terms which reduce to terms in the simple calculus, even though not all terms will reduce to simple terms. For instance the identity function $\lambda x_1.x_1$ is irreducible even though the term $(\lambda x_1.x_1\ p_1)$ reduces to p_1. I will say that two terms represent the same abstract term if and only if they reduce to two simple terms with the same abstract syntax. For this, I extend the abstract syntax function Ab on $\mathfrak{H}$ with the partial function $Ab_{\mathfrak{H}}$ from $\mathfrak{H}$ to Syn such that

$$Ab_{\mathfrak{H}}[\![\mathcal{T}]\!] = \begin{cases} Ab[\![\mathcal{S}]\!] & \exists \mathcal{S} \in \mathfrak{S}.\ \mathcal{T} \twoheadrightarrow_{\beta\delta} \mathcal{S} \\ \text{undefined} & \text{otherwise} \end{cases}$$

While this is a natural extension of abstract syntax to the extended calculus, it is no longer the case that $Ab_{\mathfrak{H}}[\![\mathcal{T}]\!] = Ab_{\mathfrak{H}}[\![\mathcal{S}]\!]$ if and only if $\mathcal{T} =_{\mathfrak{H}} \mathcal{S}$. This is a result of the more expressive syntax for the extended calculus. The property that extends to this calculus is this slightly weaker result:

Theorem 2.6. *For any terms $\mathcal{T}, \mathcal{S} \in \mathfrak{H}$, if $Ab_{\mathfrak{H}}[\![\mathcal{T}]\!]$ and $Ab_{\mathfrak{H}}[\![\mathcal{S}]\!]$ are defined, then $\mathcal{T} =_{\mathfrak{H}} \mathcal{S}$ if and only if $Ab_{\mathfrak{H}}[\![\mathcal{T}]\!] = Ab_{\mathfrak{H}}[\![\mathcal{S}]\!]$.*

Proof. If $Ab_{\mathfrak{H}}[\![\mathcal{T}]\!]$ and $Ab_{\mathfrak{H}}[\![\mathcal{S}]\!]$ are defined, then there exist terms $\mathcal{T}', \mathcal{S}' \in \mathfrak{S}$ with $\mathcal{T} \twoheadrightarrow_{\beta\delta} \mathcal{T}'$ and $\mathcal{S} \twoheadrightarrow_{\beta\delta} \mathcal{S}'$. If $\mathcal{T} =_{\mathfrak{H}} \mathcal{S}$, then by the Church Rosser property, $\mathcal{T}' \equiv \mathcal{S}'$, so $Ab_{\mathfrak{H}}[\![\mathcal{T}]\!] = Ab_{\mathfrak{H}}[\![\mathcal{S}]\!]$. If $Ab_{\mathfrak{H}}[\![\mathcal{T}]\!] = Ab_{\mathfrak{H}}[\![\mathcal{S}]\!]$, then $\mathcal{T}' \equiv \mathcal{S}'$ by the Church Rosser property, so $\mathcal{T} =_{\mathfrak{H}} \mathcal{S}$. □

2.3 Conclusions

In this, chapter I develop the simple calculus, which serves as a framework for composition languages without higher-order components. This calculus comes equipped with a notion of structural equivalence, which is provably decidable. Two terms which are structurally equivalent have the same abstract syntax using the definition I provide.

By augmenting this with functional abstraction, I provide an extended calculus which serves as a framework for higher-order composition languages. This extended calculus has the ability to represent terms drastically more succinctly than in the simple calculus. After defining the size of terms in the calculi, I prove there a set of terms in the simple calculus whose size must grow with $\Omega(2^n)$, or $\Omega(2^{2^n})$, or $\Omega(2^{2^{2^n}})$, and so on, and an equivalent set of terms in the extended calculus whose size only need only grow with $O(n)$. This result suggests that higher-order composition languages can provide much more scalable methods to represent structures than composition languages without higher-order components. The downside of higher-order composition languages is that testing structural equivalence between terms in such a language

becomes undecidable, which is the direct result of its more expressive syntax.

Chapter 3

The Extended Calculus as a Semantic Domain

In this chapter, I map a simple higher-order composition language for composing boolean circuits to the extended calculus of the previous chapter. I use this mapping to then prove structural equivalence between components in the language. In this sense, the extended calculus serves as a semantic domain for the more practical language. This idea was inspired by Milner [43], who used his calculus of communicating systems as a semantic domain for a simple parallel programming language. I note that it is possible to use the extended calculus as a semantic domain for Ptalon as well, but the bookkeeping involved becomes too burdensome for it to be worth the effort.

3.1 The Circuit Language

Let V be an infinite set of *variable names*. These can by arbitrary strings, excluding keywords like `true` and `false`. Define the set B of *boolean expressions* to be the smallest set such that

1. `true` and `false` belong to B.
2. If $v \in V$, then $v \in B$.
3. If $b_1, b_2 \in B$, then
   (a) `not` b_1
   (b) b_1 `and` b_2
   (c) b_1 `or` b_2
   (d) b_1 `xor` b_2
   (e) b_1 `nand` b_2
   (f) b_1 `nor` b_2
   (g) b_1 `xnor` b_2

   all belong to B.
4. If $b \in B$, then `(`b`)` belongs to B.

Define the set E of *boolean equations* as the smallest set such that if $v \in V$ and $b \in B$, then v `=` b belongs to E. An example in E is

```
out = in1 and (in2 or in3)
```

Define the set T of *terms* as the smallest set such that

1. `empty` belongs to T.
2. If $v \in V$, then $v \in T$.
3. If $v \in V$, then `port` v belongs to T.
4. If $v \in V$, then `wire` v belongs to T.
5. If $t \in T$, then `port p<`t`>` belongs to T.
6. If $t \in T$, then `wire r<`t`>` belongs to T.
7. If $e \in E$, then e belongs to T.
8. If $v_1, v_2, v_3 \in V$, and $t_1, t_2, t_3 \in T$, then
   (a) v_1 `{`v_2 `:=` v_3`}`
   (b) v_1`{`v_2`:= p<`t_3`>}`
   (c) v_1 `{p<`t_2`> :=` v_3`}`
   (d) v_1`{p<`t_2`> := p<`t_3`>}`
   (e) `l<`t_1`>{`v_2 `:=` v_3`}`
   (f) `l<`t_1`>{`v_2 `:= p<`t_3`>}`
   (g) `l<`t_1`>{p<`t_2`> :=` v_3`}`

(h) `l<`v_1`>{p<`v_2`> := p<`v_3`>}`

and

(a) v_1`{`v_2`:= r<`t_3`>}`

(b) v_1`{p<`t_2`> := r<`t_3`>}`

(c) `l<`t_1`>{`v_2 `:= r<`t_3`>}`

(d) `l<`v_1`>{p<`v_2`> := r<`v_3`>}`

belong to T.

9. If $t_1, t_2 \in T$, then t_1`;` t_2 belongs to T.

10. If $v \in V$ and $t \in T$, then v`:[`t`]` belongs to T.

11. If $t_1, t_2 \in T$, then `l<`t_1`>:[`t_2`]` belongs to T.

12. If $v \in V$ and $t \in T$, then `(\`v`.`T`)` belongs to T.

13. If $t_1, t_2 \in T$, then `apply(`t_1`,` t_2`)` belongs to T.

An example term that describes an AND gate is

```
port in1;
port in2;
port out;
out = in1 and in2
```

A simple composite term that creates a NAND gate from an AND gate and a NOT gate is described by

```
port in1;
port in2;
port out;

andGate:[
    port in1;
    port in2;
    port out;
    out = in1 and in2
];

notGate:[
    port in;
    port out;
    out = not in
];

relation r;
andGate(in1 := in1);
andGate(in2 := in2);
andGate(out := r);
orGate(in := r);
orGate(out := out)
```

In this syntax a term like `(\`$v.t$`)` represents an functional abstraction with a variable v. This can be viewed as a lambda abstraction. This combined with the `apply` operator gives this language the power of lambda calculus. To create simple higher-order term to compose components `x` and `y` in a parallel network, I can use the expression

```
(\x.
    (\y.
        component1:[x];
        component2:[y]
    )
)
```

To make it easier to do simple arithmetic, I provide the keywords `0`, `1`, ..., `+`, `-`, `*`, `div`, `mod`, `=`, `<=`, `>=`, `<`, `>`, `!=`, `&&`, `||`, and `!`. Each such keyword represents an appropriate lambda term. Each such term can be substituted with a term in the language above, so I do not include it in the "official" language. For instance `1` can be substituted with

```
(\f.
    (\x.
        apply(f, x)
    )
)
```

and `*` can be substituted with

```
(\m.
    (\n.
        (\f.
            apply(m, apply(n, f))
        )
    )
)
```

Then I can use

```
apply(apply(*,3), 2)
```

to represent three times two. I will take this a step further and let

```
3 * 2
```

represent the same thing. The mapping from such expressions to λ terms is routine, so I omit the details. I also assume the keywords `T`, `F`, and `Y` to represent true, false, and the Y combinator of Equation 2.4 respectively. I similarly employ an `if-else` construct with

```
if x {
    y
} else {
    z
}
```

equivalent to

```
apply(apply(x, y), z)
```

Replacing `x` with `T` or `F` then makes this behave as expected. Finally, I will let an expression like

```
apply(f, x, y, z)
```

be shorthand for

```
apply(apply(apply(f, x), y), z)
```

using the same parenthesis rule as function application in λ calculus.

With these terms, I can now define a high-order structure that duplicates n copies of term x in parallel in a relatively straightforward manner.

```
apply(Y,
    (\f.
        (\n.
            (\x.
                if (n == 0) {
                    empty
                } else {
                    l<n>:[x];
                    apply(f, n-1, x)
                }
            )
        )
    )
)
```

Note here the use of the `<>` construct. Its purpose is to return a unique name for each number. Here, `<0>` returns `_0`, `<1>` returns `_1`, and so on. This gets concatenated to `l`, as in `l_0`, `l_1`, and so on. In this case, each component in parallel receives a new label.

3.2 Intermediate Language

Note that programs written in the above language might have a problem with overlapping names. For instance, the following is an *invalid program*:

```
l:[
    port in;
]

port p;
relation p;
l{in := p}
```

It's not clear if the port `in` is connected to the port `p` or the relation `r`. Since these sorts of errors can easily be caught with a scope checker, this is no big deal in practice. In the semantic mapping, however, this means I have to worry about a "scope environment," as is required in the study of imperative programming languages [52].

To avoid this extra complication I develop an *intermediate language* for the semantic study. It is almost identical to the above language except that I am more explicit about the "name space" of terms. Assume that X, R, P, and L are the mutually disjoint and infinite subsets of V corresponding to variables names, relation names,

port names, and component label names respectively. I will assume the only names in these sets are `x_0`, `x_1`, ... $\in$ `X`, `r_0`, `r_1`, ... $\in$ `R`, `p_0`, `p_1`, ... $\in$ `P`, and `l_0`, `l_1`, ... $\in$ `L`.

In this intermediate language I redefine the sets B, E, and T as follows: Define the set B of *boolean expressions* to be the smallest set such that

1. `true` and `false` belong to B.
2. If `x_`$i \in$ `X`, then `x_`$i \in$ `B`.
3. If $b_1, b_2 \in B$, then
   (a) `not` b_1
   (b) b_1 `and` b_2
   (c) b_1 `or` b_2
   (d) b_1 `xor` b_2
   (e) b_1 `nand` b_2
   (f) b_1 `nor` b_2
   (g) b_1 `xnor` b_2

   all belong to B.
4. If $b \in B$, then `(`b`)` belongs to B.

Define the set E of *boolean equations* as the smallest set such that if `x_`$i \in$ `X` and $b \in B$, then `x_`i `=` b belongs to E.

Define the set T of *terms* as the smallest set such that

1. `empty` belongs to T.

2. If `x_`$i \in$ `X`, then `x_`$i \in$ `T`.

3. If `p_`$i \in$ `P`, then `port p_`i belongs to T.

4. If `r_`$i \in$ `R`, then `wire r_`i belongs to T.

5. If $t \in T$, then `port p<`t`>` belongs to T.

6. If $t \in T$, then `wire r<`t`>` belongs to T.

7. If $e \in E$, then e belongs to T.

8. If `l_`$i \in$ `L`, `p_`$j \in$ `P`, `r_`$k \in$ `R`, and $t_1, t_2, t_3 \in T$, then

   (a) `l_`i `{p_`j `:=` `p_`k`}`

   (b) `l_`i`{p_`j`:=` `p<`t_3`>}`

   (c) `l_`i `{p<`t_2`>` `:=` `p_`k`}`

   (d) `l_`i`{p<`t_2`>` `:=` `p<`t_3`>}`

   (e) `l<`t_1`>{p_`j `:=` `p_`k`}`

   (f) `l<`t_1`>{p_`j `:=` `p<`t_3`>}`

   (g) `l<`t_1`>{p<`t_2`>` `:=` `p_`k`}`

   (h) `l<`t_1`>{p<`t_2`>` `:=` `p<`t_3`>}`

   and

(a) `l_`i`{p_`j`:= r<`t_3`>}`

(b) `l_`i`{p<`t_2`> := r<`t_3`>}`

(c) `l<`t_1`>{p_`j` := r<`t_3`>}`

(d) `l<`t_1`>{p<`t_2`> := r<`t_3`>}`

belong to T.

9. If $t_1, t_2 \in T$, then t_1`;` t_2 belongs to T.

10. If `l_`$i \in$ `L` and $t \in T$, then `l_`i`:[`t`]` belongs to T.

11. If $t_1, t_2 \in T$, then `l<`t_1`>:[`t_2`]` belongs to T.

12. If `x_`$i \in$ `V` and $t \in T$, then `(\x_`i`.`T`)` belongs to T.

13. If $t_1, t_2 \in T$, then `apply(`t_1`,` t_2`)` belongs to T.

3.3 Semantics of the Circuit Language

For each boolean expression $b \in B$, I will define a number $\mathfrak{G}[\![b]\!]$ as follows:

$$\begin{aligned}
\mathfrak{G}[\![\texttt{true}]\!] &= 1 \\
\mathfrak{G}[\![\texttt{false}]\!] &= 2 \\
\mathfrak{G}[\![x_i]\!] &= 3^i \\
\mathfrak{G}[\![\texttt{notb}]\!] &= 5^{\mathfrak{G}[\![b]\!]} \\
\mathfrak{G}[\![b_1 \texttt{ and b}_2]\!] &= 7^{\mathfrak{G}[\![b_1]\!]} \cdot 11^{\mathfrak{G}[\![b_2]\!]} \\
\mathfrak{G}[\![b_1 \texttt{ or b}_2]\!] &= 13^{\mathfrak{G}[\![b_1]\!]} \cdot 17^{\mathfrak{G}[\![b_2]\!]} \\
\mathfrak{G}[\![b_1 \texttt{ xor b}_2]\!] &= 19^{\mathfrak{G}[\![b_1]\!]} \cdot 23^{\mathfrak{G}[\![b_2]\!]} \\
\mathfrak{G}[\![b_1 \texttt{ nand b}_2]\!] &= 29^{\mathfrak{G}[\![b_1]\!]} \cdot 31^{\mathfrak{G}[\![b_2]\!]} \\
\mathfrak{G}[\![b_1 \texttt{ nor b}_2]\!] &= 41^{\mathfrak{G}[\![b_1]\!]} \cdot 43^{\mathfrak{G}[\![b_2]\!]} \\
\mathfrak{G}[\![b_1 \texttt{ xnor b}_2]\!] &= 47^{\mathfrak{G}[\![b_1]\!]} \cdot 53^{\mathfrak{G}[\![b_2]\!]}
\end{aligned}$$

This is a *Gödel numbering system*, similar to that in [12]. Since prime numbers are used here, it is easy to check that each expression $b \in B$ has a unique Gödel number. I can extend this to number boolean equations $e \in E$ with

$$\mathfrak{G}[\![x_i = \texttt{b}]\!] = 59^{\mathfrak{G}[\![\texttt{x}_\texttt{i}]\!]} \cdot 61^{\mathfrak{G}[\![\texttt{b}]\!]}$$

I am now ready to define the semantic function $\mathfrak{D} : T \to \mathfrak{H}$.

$$
\begin{aligned}
\mathfrak{D}[\![\texttt{empty}]\!] &= 0 \\
\mathfrak{D}[\![\texttt{x\_}i]\!] &= x_i \\
\mathfrak{D}[\![\texttt{p\_}i]\!] &= p_i \\
\mathfrak{D}[\![\texttt{port p} <t>]\!] &= p_{\langle \mathfrak{D}[\![t]\!] \rangle} \\
\mathfrak{D}[\![\texttt{relation r\_}i]\!] &= r_i \\
\mathfrak{D}[\![\texttt{relation r} <t>]\!] &= r_{\langle \mathfrak{D}[\![t]\!] \rangle} \\
\mathfrak{D}[\![e]\!] &= A_{\mathfrak{G}[\![e]\!]} \\
\mathfrak{D}[\![\texttt{l\_}i\{\texttt{p\_}j := \texttt{p\_}k\}]\!] &= l_i\{p_j := p_k\} \\
\mathfrak{D}[\![\texttt{l\_}i\{\texttt{p\_}j := \texttt{p} <t>\}]\!] &= l_i\{p_j := p_{\langle \mathfrak{D}[\![t]\!] \rangle}\} \\
\mathfrak{D}[\![\texttt{l\_}i\{\texttt{p} <t> := \texttt{p\_}k\}]\!] &= l_i\{p_{\langle \mathfrak{D}[\![t]\!] \rangle} := p_k\} \\
\mathfrak{D}[\![\texttt{l\_}i\{\texttt{p} <t_1> := \texttt{p} <t_2>\}]\!] &= l_i\{p_{\langle \mathfrak{D}[\![t_1]\!] \rangle} := p_{\langle \mathfrak{D}[\![t_2]\!] \rangle}\} \\
\mathfrak{D}[\![\texttt{l} <t> \{\texttt{p\_}j := \texttt{p\_}k\}]\!] &= l_{\langle \mathfrak{D}[\![t]\!] \rangle}\{p_j := p_k\} \\
\mathfrak{D}[\![\texttt{l} <t_1> \{\texttt{p\_}j := \texttt{p} <t_2>\}]\!] &= l_{\langle \mathfrak{D}[\![t_1]\!] \rangle}\{p_j := p_{\langle \mathfrak{D}[\![t_2]\!] \rangle}\} \\
\mathfrak{D}[\![\texttt{l} <t_1> \{\texttt{p} <t_2> := \texttt{p\_}k\}]\!] &= l_{\langle \mathfrak{D}[\![t_1]\!] \rangle}\{p_{\langle \mathfrak{D}[\![t_2]\!] \rangle} := p_k\} \\
\mathfrak{D}[\![\texttt{l} <t_1> \{\texttt{p} <t_2> := \texttt{p} <t_3>\}]\!] &= l_{\langle \mathfrak{D}[\![t_1]\!] \rangle}\{p_{\langle \mathfrak{D}[\![t_2]\!] \rangle} := p_{\langle \mathfrak{D}[\![t_3]\!] \rangle}\}
\end{aligned}
$$

and

$$
\begin{aligned}
\mathfrak{D}[\![\texttt{l\_}i\{\texttt{p\_}j := \texttt{r\_}k\}]\!] &= l_i\{p_j := r_k\} \\
\mathfrak{D}[\![\texttt{l\_}i\{\texttt{p\_}j := \texttt{r} <t>\}]\!] &= l_i\{p_j := r_{\langle \mathfrak{D}[\![t]\!] \rangle}\} \\
\mathfrak{D}[\![\texttt{l\_}i\{\texttt{p} <t> := \texttt{r\_}k\}]\!] &= l_i\{p_{\langle \mathfrak{D}[\![t]\!] \rangle} := r_k\} \\
\mathfrak{D}[\![\texttt{l\_}i\{\texttt{p} <t_1> := \texttt{r} <t_2>\}]\!] &= l_i\{p_{\langle \mathfrak{D}[\![t_1]\!] \rangle} := r_{\langle \mathfrak{D}[\![t_2]\!] \rangle}\} \\
\mathfrak{D}[\![\texttt{l} <t> \{\texttt{p\_}j := \texttt{r\_}k\}]\!] &= l_{\langle \mathfrak{D}[\![t]\!] \rangle}\{p_j := r_k\} \\
\mathfrak{D}[\![\texttt{l} <t_1> \{\texttt{p\_}j := \texttt{r} <t_2>\}]\!] &= l_{\langle \mathfrak{D}[\![t_1]\!] \rangle}\{p_j := r_{\langle \mathfrak{D}[\![t_2]\!] \rangle}\} \\
\mathfrak{D}[\![\texttt{l} <t_1> \{\texttt{p} <t_2> := \texttt{r\_}k\}]\!] &= l_{\langle \mathfrak{D}[\![t_1]\!] \rangle}\{p_{\langle \mathfrak{D}[\![t_2]\!] \rangle} := r_k\} \\
\mathfrak{D}[\![\texttt{l} <t_1> \{\texttt{p} <t_2> := \texttt{r} <t_3>\}]\!] &= l_{\langle \mathfrak{D}[\![t_1]\!] \rangle}\{p_{\langle \mathfrak{D}[\![t_2]\!] \rangle} := r_{\langle \mathfrak{D}[\![t_3]\!] \rangle}\} \\
\mathfrak{D}[\![t_1;\ t_2]\!] &= \mathfrak{D}[\![t_1]\!] \oplus \mathfrak{D}[\![t_2]\!] \\
\mathfrak{D}[\![\ \texttt{l\_}i : [t]\ ]\!] &= l_i : [\ \mathfrak{D}[\![t]\!]\ ] \\
\mathfrak{D}[\![\ \texttt{l} <t_1> : [t_2]\ ]\!] &= l_{\langle \mathfrak{D}[\![t_1]\!] \rangle} : [\ \mathfrak{D}[\![t_2]\!]\ ] \\
\mathfrak{D}[\![(\backslash \texttt{x\_}i.t)]\!] &= \lambda x_i.\mathfrak{D}[\![t]\!] \\
\mathfrak{D}[\![\texttt{apply}(t_1, t_2)]\!] &= (\mathfrak{D}[\![t_1]\!]\ \mathfrak{D}[\![t_2]\!])
\end{aligned}
$$

3.4 Proofs Using the Semantics Function

I now show how the semantics function and the structural equivalence $=_{\mathfrak{H}}$ can be used to prove that two components in the circuit language are the same.

3.4.1 Parallel Composition

Consider again the parallel composition system, this time written in the intermediate language, but I still use a few abbreviations, since they have direct mappings to λ calculus.

```
apply(Y,
    (\x_1.
        (\x_2.
            (\x_3.
                if (x_2 == 0) {
                    empty
                } else {
                    l<x_2>:[x_3];
                    apply(x_1, x_2-1, x_3)
                }
            )
        )
    )
)
```

I will prove that

```
apply(
    (\x_1.
        (\x_2.
            (\x_3.
                if (x_2 == 0) {
                    empty
                } else {
                    l<x_2>:[x_3];
                    apply(x_1, x_2-1, x_3)
                }
            )
        )
    ),
    3,
    port p_1
)
```

is structurally equivalent to

```
l_1:[port p_1];
l_2:[port p_1];
l_3:[port p_1]
```

Note that the denotation of the first program is

$$\mathcal{S}_1 \equiv_{def} (\mathcal{Y}_{fp}\ \lambda x_1.\lambda x_2.\lambda x_3.(\mathcal{F}_{zero}\ x_2)\ 0\ (l_{\langle x_2\rangle}:[x_3] \oplus (x_1\ (\mathcal{F}_{pred}\ x_2)\ x_3)))\ \mathcal{C}_3\ p_1$$

The second term has denotation

$$\mathcal{S}_2 \equiv_{def} l_1:[p_1] \oplus l_2:[p_1] \oplus l_2:[p_1]$$

If $\mathcal{S}_1 =_{\mathfrak{H}} \mathcal{S}_2$, then they are structurally equivalent. Note that

$$\begin{aligned}
\mathcal{S}_1 \quad &\twoheadrightarrow_{\beta\delta} \quad (\lambda x_1.\lambda x_2.\lambda x_3.(\mathcal{F}_{zero}\ x_2)\ 0\ (l_{\langle x_2\rangle}:[x_3] \oplus (x_1\ (\mathcal{F}_{pred}\ x_2)\ x_3) \\
&\qquad (\mathcal{Y}_{fp}\ \lambda x_1.\lambda x_2.\lambda x_3.(\mathcal{F}_{zero}\ x_2)\ 0\ (l_{\langle x_2\rangle}:[x_3] \oplus (x_1\ (\mathcal{F}_{pred}\ x_2)\ x_3))))\ \mathcal{C}_3\ p_1 \\
&\twoheadrightarrow_{\beta\delta} \quad l_{\langle C_3\rangle}:[p_1] \oplus \\
&\qquad ((\mathcal{Y}_{fp}\ \lambda x_1.\lambda x_2.\lambda x_3.(\mathcal{F}_{zero}\ x_2)\ 0\ (l_{\langle x_2\rangle}:[x_3] \oplus (x_1\ (\mathcal{F}_{pred}\ x_2)\ x_3)))\ \mathcal{C}_2\ p_1) \\
&\twoheadrightarrow_{\beta\delta} \quad l_3:[p_1] \oplus l_2:[p_1] \oplus \\
&\qquad ((\mathcal{Y}_{fp}\ \lambda x_1.\lambda x_2.\lambda x_3.(\mathcal{F}_{zero}\ x_2)\ 0\ (l_{\langle x_2\rangle}:[x_3] \oplus (x_1\ (\mathcal{F}_{pred}\ x_2)\ x_3)))\ \mathcal{C}_1\ p_1) \\
&\twoheadrightarrow_{\beta\delta} \quad l_3:[p_1] \oplus l_2:[p_1] \oplus l_1:[p_1] \oplus \\
&\qquad ((\mathcal{Y}_{fp}\ \lambda x_1.\lambda x_2.\lambda x_3.(\mathcal{F}_{zero}\ x_2)\ 0\ (l_{\langle x_2\rangle}:[x_3] \oplus (x_1\ (\mathcal{F}_{pred}\ x_2)\ x_3)))\ \mathcal{C}_0\ p_1) \\
&\twoheadrightarrow_{\beta\delta} \quad l_3:[p_1] \oplus l_2:[p_1] \oplus l_1:[p_1] \oplus 0
\end{aligned}$$

Now

$$\begin{aligned}
\mathcal{T}_1 &\equiv_{def} l_3 : [p_1] \oplus l_2 : [p_1] \oplus l_1 : [p_1] \oplus 0 \\
&=_{\mathfrak{S}} l_3 : [p_1] \oplus l_2 : [p_1] \oplus l_1 : [p_1] \\
&=_{\mathfrak{S}} l_1 : [p_1] \oplus l_2 : [p_1] \oplus l_2 : [p_1] \\
&\equiv \mathcal{S}_2
\end{aligned}$$

Since $\mathcal{S}_1 \twoheadrightarrow_{\beta\delta} \mathcal{T}_1$, and $\mathcal{T}_1 =_{\mathfrak{S}} \mathcal{S}_2$, $\mathcal{S}_1 =_{\mathfrak{H}} \mathcal{S}_2$.

3.4.2 Adder Circuit

A full adder can be written in the intermediate language as

```
port p_1;
port p_2;
port p_3;
port p_4;
port p_5;
p_4 = (p_1 xor p_2) xor p_3;
p_5 = ((p_1 xor p_2) and p_3) or (p_1 and p_2)
```

Here `p_1` and `p_2` are the input bits, `p_3` is the carry in bit, `p_4` is the sum bit, and `p_5` is the carry bit.

A two-bit adder in the intermediate language, made from basic adders can be written as

```
port p_1;
port p_2;
port p_3;
port p_4;
port p_5;
port p_6;
```

```
port p_7;
port p_8;

l_1:[
    port p_1;
    port p_2;
    port p_3;
    port p_4;
    port p_5;
    p_4 = (p_1 xor p_2) xor p_3;
    p_5 = ((p_1 xor p_2) and p_3) or (p_1 and p_2)
]

l_2:[
    port p_1;
    port p_2;
    port p_3;
    port p_4;
    port p_5;
    p_4 = (p_1 xor p_2) xor p_3;
    p_5 = ((p_1 xor p_2) and p_3) or (p_1 and p_2)
]

wire r_1;

l_1{p_1 := p_1}; l_1{p_2 := p_3}; l_1{p_3 := p_5}; l_1{p_4 := p_6};
l_1{p_4 := r_1};

l_2{p_1 := p_2}; l_2{p_2 := p_4}; l_2{p_3 := r_1}; l_2{p_4 := p_7};
l_2{p_5 := p_8};
```

Here, `p_1` and `p_3` are the two bits from the first input, `p_2` and `p_4` are the two bits from the second input, `p_5` is the carry in bit, `p_6` and `p_7` are the sum bits, and `p_8` is the carry out bit.

To make a higher-order component to create an n bit adder in the intermediate language, assuming $x_1 \geq 2$, I use:

```
(\x_1.
```

```
apply(
    apply(Y,
        (\x_2.(\x_3.
            if (x_3 == 0) {
                empty
            } else {
                p<x_3>;
                apply(x_2, x_3-1)
            }
        ))
    ),
    2 + 3*x_1
);
apply(
    apply(Y,
        (\x_2.(\x_3.
            if (x_3 == 0) {
                empty
            } else {
                l<x_3>:[
                    port p_1;
                    port p_2;
                    port p_3;
                    port p_4;
                    port p_5;
                    p_4 = (p_1 xor p_2) xor p_3;
                    p_5 = ((p_1 xor p_2) and p_3) or (p_1 and p_2)
                ]
                apply(x_2, x_3-1)
            }
        ))
    ),
    x_1 - 2
);
apply(
    apply(Y,
        (\x_2.(\x_3.
            if (x_3 == 0) {
                empty
            } else {
                r<x_3>;
                apply(x_2, x_3-1)
```

```
                }
            ))
        ),
        x_1 - 1
    );
    l_1{p_1 := p_1};
    l_1{p_2 := p<1 + x_1>};
    l_1{p_3 := p<1 + 2*x_1>};
    l_1{p_4 := p<2 + 2*x_1>};
    l_1{p_5 := r_1};
    apply(
        apply(Y,
            (\x_2.(\x_3.
                if (x_3 == 0) {
                    empty
                } else {
                    l<x_3>{p_1 := p<1 + x_3>};
                    l<x_3>{p_2 := p<1 + x_3 + x_1>};
                    l<x_3>{p_3 := r<x_3>};
                    l<x_3>{p_4 := p<2 + x_3 + 2*x_1>};
                    l<x_3>{p_5 := r<x_3 + 1>};
                    apply(x_2, x_3-1)
                }
            ))
        ),
        x_1 - 2
    );
    l<x_1>{p_1 := p<x_1>};
    l<x_1>{p_2 := p<2*x_1>};
    l<x_1>{p_3 := r<x_1-1>};
    l<x_1>{p_4 := p<1 + 3*x_1>};
    l<x_1>{p_5 := p<2 + 3*x_1>};
)
```

The denotation for this higher-order component is given by

$$\mathcal{S} \equiv_{def} \lambda x_1.(\mathcal{S}_1 \oplus \mathcal{S}_2 \oplus \mathcal{S}_3 \oplus \mathcal{S}_4 \oplus \mathcal{S}_5 \oplus \mathcal{S}_6)$$

with

$$\mathcal{S}_1 \equiv_{def} ((\mathcal{Y}_{fp}\ (\lambda x_2.\lambda x_3.(\mathcal{F}_{zero}\ x_3)\ 0\ (p_{\langle x_3\rangle} \oplus (x_2\ (\mathcal{F}_{pred}\ x_3)))))\ (\mathcal{F}_+\ \mathcal{C}_2\ (\mathcal{F}_*\ \mathcal{C}_3\ x_1)))$$

$$\mathcal{S}_2 \equiv_{def} ((\mathcal{Y}_{fp}\ (\lambda x_2.\lambda x_3.(\mathcal{F}_{zero}\ x_3)\ 0\ (\mathcal{S}_7 \oplus (x_2\ (\mathcal{F}_{pred}\ x_3)))))\ (\mathcal{F}_-\ x_1\ \mathcal{C}_2))$$

$$\mathcal{S}_3 \equiv_{def} ((\mathcal{Y}_{fp}\ (\lambda x_2.\lambda x_3.(\mathcal{F}_{zero}\ x_3)\ 0\ (r_{\langle x_3\rangle} \oplus (x_2\ (\mathcal{F}_{pred}\ x_3)))))\ (\mathcal{F}_{pred}\ x_1))$$

$$\mathcal{S}_4 \equiv_{def} l_1\{p_1 := p_1\} \oplus l_1\{p_2 := p_{\langle \mathcal{F}_+\ \mathcal{C}_1\ x_1\rangle}\} \oplus l_1\{p_3 := p_{\langle \mathcal{F}_+\ \mathcal{C}_1\ (\mathcal{F}_{pred}\ x_1)\rangle}\}$$
$$\oplus\ l_1\{p_4 := p_{\langle \mathcal{F}_+\ \mathcal{C}_2\ (\mathcal{F}_*\ \mathcal{C}_2\ x_1)\rangle}\} \oplus l_1\{p_5 := r_1\}$$

$$\mathcal{S}_5 \equiv_{def} ((\mathcal{Y}_{fp}\ (\lambda x_2.\lambda x_3.(\mathcal{F}_{zero}\ x_3)\ 0\ (\mathcal{S}_8 \oplus (x_2\ (\mathcal{F}_{pred}\ x_3)))))\ (\mathcal{F}_-\ x_1\ \mathcal{C}_2))$$

$$\mathcal{S}_6 \equiv_{def} l_{\langle x_1\rangle}\{p_1 := p_{\langle x_1\rangle}\} \oplus l_{\langle x_1\rangle}\{p_2 := p_{\langle \mathcal{F}_*\ \mathcal{C}_2\ x_1\rangle}\} \oplus l_{\langle x_1\rangle}\{p_3 := r_{\langle \mathcal{F}_{pred}\ x_1\rangle}\}$$
$$\oplus\ l_{\langle x_1\rangle}\{p_4 := p_{\langle \mathcal{F}_+\ \mathcal{C}_1\ (\mathcal{F}_*\ \mathcal{C}_3\ x_1)\rangle}\} \oplus l_{\langle x_1\rangle}\{p_5 := p_{\langle \mathcal{F}_+\ \mathcal{C}_2\ (\mathcal{F}_*\ \mathcal{C}_3\ x_1)\rangle}\}$$

$$\mathcal{S}_7 \equiv_{def} l_{\langle x_3\rangle} : [p_1 \oplus p_2 \oplus p_3 \oplus p_4 \oplus p_5 \oplus A_{n_1} \oplus A_{n_2}]$$

$$\mathcal{S}_8 \equiv_{def} l_{\langle x_1\rangle}\{p_1 := p_{\langle \mathcal{F}_+\ \mathcal{C}_1\ x_1\rangle}\} \oplus l_{\langle x_1\rangle}\{p_2 := p_{\langle \mathcal{F}_+\ (\mathcal{F}_+\ \mathcal{C}_1\ x_3)\ x_1\rangle}\} \oplus l_{\langle x_1\rangle}\{p_3 := r_{\langle x_3\rangle}\}$$
$$\oplus\ l_{\langle x_1\rangle}\{p_4 := p_{\langle \mathcal{F}_+\ (\mathcal{F}_+\ \mathcal{C}_2\ x_3)\ (\mathcal{F}_*\ \mathcal{C}_2\ x_1)\rangle}\} \oplus l_{\langle x_1\rangle}\{p_5 := r_{\langle \mathcal{F}_+\ \mathcal{C}_1\ x_1\rangle}\}$$

where

$$n_1 = \mathfrak{G}[\![\mathtt{p_4 = (p_1\ xor\ p_2)\ xor\ p_3}]\!]$$

$$n_2 = \mathfrak{G}[\![\mathtt{p_5 = ((p_1\ xor\ p_2)\ and\ p_3)\ or\ (p_1\ and\ p_2)}]\!]$$

I would like to prove that $(\mathcal{S}\ \mathcal{C}_2)$ is equivalent to the denotation for the two-bit adder, which is given by

$$\mathcal{T} \equiv_{def} \lambda x_1.(\mathcal{T}_1 \oplus \mathcal{T}_2 \oplus \mathcal{T}_3 \oplus \mathcal{T}_4 \oplus \mathcal{T}_5)$$

with

$$\begin{aligned}
\mathcal{T}_1 &\equiv_{def} p_1 \oplus p_2 \oplus p_3 \oplus p_4 \oplus p_5 \oplus p_6 \oplus p_7 \oplus p_8 \\
\mathcal{T}_2 &\equiv_{def} l_1 : [p_1 \oplus p_2 \oplus p_3 \oplus p_4 \oplus p_5 \oplus A_{n_1} \oplus A_{n_2}] \oplus \\
&\qquad l_2 : [p_1 \oplus p_2 \oplus p_3 \oplus p_4 \oplus p_5 \oplus A_{n_1} \oplus A_{n_2}] \\
\mathcal{T}_3 &\equiv_{def} r_1 \oplus r_2 \oplus r_3 \oplus r_4 \oplus r_5 \oplus r_6 \oplus r_7 \oplus r_8 \\
\mathcal{T}_4 &\equiv_{def} l_1\{p_1 := p_1\} \oplus l_1\{p_2 := p_3\} \oplus l_1\{p_3 := p_5\} \oplus l_1\{p_4 := p_6\} \oplus l_1\{p_5 := r_1\} \\
\mathcal{T}_5 &\equiv_{def} l_2\{p_1 := p_2\} \oplus l_2\{p_2 := p_4\} \oplus l_2\{p_3 := r_1\} \oplus l_2\{p_4 := p_7\} \oplus l_2\{p_5 := p_8\}
\end{aligned}$$

It is easy to check that

$$\begin{aligned}
\mathcal{S}_1\lfloor \mathcal{C}_2/x_1 \rfloor &=_{\mathfrak{H}} \mathcal{T}_1 \\
\mathcal{S}_2\lfloor \mathcal{C}_2/x_1 \rfloor &=_{\mathfrak{H}} \mathcal{T}_2 \\
\mathcal{S}_3\lfloor \mathcal{C}_2/x_1 \rfloor &=_{\mathfrak{H}} \mathcal{T}_3 \\
\mathcal{S}_4\lfloor \mathcal{C}_2/x_1 \rfloor &=_{\mathfrak{H}} \mathcal{T}_4 \\
\mathcal{S}_5\lfloor \mathcal{C}_2/x_1 \rfloor &=_{\mathfrak{H}} 0 \\
\mathcal{S}_6\lfloor \mathcal{C}_2/x_1 \rfloor &=_{\mathfrak{H}} \mathcal{T}_5
\end{aligned}$$

From this, it is straightforward to conclude that $(\mathcal{S}\ \mathcal{C}_2) =_{\mathfrak{H}} \mathcal{T}$.

3.4.3 Scalability of a Clock Distribution Network

In integrated circuit design, minimizing clock skew and the rise and fall times of a clock source is important to provide precision system timing [33]. Clock distribution networks use stages of buffers to fan out a clock source to multiple components while minimizing the effects of clock skew and the rise and fall times of a clock source. A very simple clock distribution network using buffers is shown in Figure 3.1. Here the clock is the input, and the outputs are "copies" of the clock value.

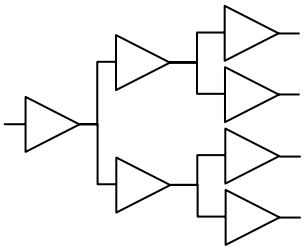

Figure 3.1: A three-level buffered clock distribution network.

Note that a one-level clock distribution network can be written as a simple buffer in the clock language, with `p_1` as the clock input:

```
port p_1;
port p_2;
p_1 = p_2
```

A two-level clock distribution now requires three buffers:

```
port p_1;
port p_2;
port p_3;

l_1 : [
```

```
    port p_1;
    port p_2;
    p_1 = p_2
];

l_1 : [
    port p_1;
    port p_2;
    p_1 = p_2
];

l_2 : [
    port p_1;
    port p_2;
    p_1 = p_2
];

l_3 : [
    port p_1;
    port p_2;
    p_1 = p_2
];

wire r_1;
l_1{p_1 := p_1};
l_1{p_2 := r_1};
l_2{p_1 := r_1};
l_2(p_2 := p_2};
l_3{p_1 := r_1};
l_3{p_2 := p_3}
```

I can go on to create three-level networks, four-level networks, and so on, all without using higher-order components. If I map these components to the simple calculus, I can prove that the denotation of an n-level clock distribution network created this way is minimal with respect to the simple calculus, and that the size of these terms grows exponentially with n. I omit these details, because this is probably easier to

infer from Figure 3.1 than from the semantic mapping.

This is where a higher-order component is useful. Let `x_0` represent the number of levels of a clock distribution network. The following higher-order component represents an `x_0`-level clock distribution network:

```
(\x_0.
    apply(
        apply(Y,
            (\x_1.
                (\x_2.
                    if (x_2 == 0) {
                        empty;
                    } else {
                        l<x_2> : [
                            port p_1;
                            port p_2;
                            p_1 = p_2
                        ];
                    }
                )
            )
        ),
        2^x_0 - 1
    )

    apply(
        apply(Y,
            (\x_1.
                (\x_2.
                    if (x_2 == 0) {
                        empty;
                    } else {
                        wire r<x_2>;
                        l<2^(x_0 - 1) + x_2>{p_2 := r<x_2>};
                        l<2*x_2 - 1>{p_1 := r<x_2>};
                        l<2*x_2>{p_1 := r<x_2>};
                    }
                )
            )
```

```
        ),
        2^(x_0 - 1) - 1
    )

    apply(
        apply(Y,
            (\x_1.
                (\x_2.
                    if (x_2 == 0) {
                        empty;
                    } else {
                        port p<x_2 + 1>;
                        l<x_2>{p_2 := p<x_2 + 1>};
                    }
                )
            )
        ),
        2^(x_0 - 1)
    )

    port p_1;
    l<2^(x_0) - 1>{p_1 := p_1}
)
```

A proof similar to the proof of the adder will show that this is indeed the correct component. Moreover, the corresponding terms in the calculus for applying this higher-order component to n will only grow linearly with n (because the representation of n in the calculus will grow linearly with n).

3.5 Conclusions

This chapter demonstrates that the extended calculus can be used as a semantic domain for a higher-order composition language. The calculus can then be used to

prove equivalence between components in the higher-order composition language. The language I chose is simple, but it demonstrates that the extended calculus does serve as a valid mathematical framework for composition languages. It is also clear that practical models in this language can scale much better when higher-order components are used, as the clock distribution network example shows.

Chapter 4

The Ptalon Programming Language

Ptalon is a higher-order composition language I developed to make it easier to create higher-order components in Ptolemy II. Ptalon is built on top of the Ptolemy Expression Language [8], a purely functional language that provides support for many data types, including arrays, matrices, and higher-order functions. Parameters can given actual values in through instantiation in other Ptalon source files, or they can be set by a user in the Ptolemy II GUI Vergil. An interpreter translates the code into component instances.

4.1 The Basics of Ptalon

Consider a simple Ptalon program:

```
Identity is {
    inport a;
    outport b;
    this(a := b);
}
```

This program defines a Ptolemy II actor.[1] Actors defined in Ptalon are generally *composite*; they are built from networks of other actors. The most basic actors in Ptolemy are *atomic*. Atomic actors are typically defined in Java code, but other languages like Python, C, and CAL [21] may also be used to define atomic actors.

The previous example is a rather trivial actor. It has two ports, an input port named a and an output port named b. That means that data flows into port a and out of port b. The keywords `inport` and `outport` are used to denote this flow direction. There is also a `port` construct in Ptalon that allows data to flow both into and out of the same port. The keyword `this` refers to the actor being created. Here the reference is used to connect the a and b ports of the Identity actor. The `:=` operator is used as a connection operator in this context. On the outside this actor would be named Identity and would have two ports a and b. On the inside this actor would connect the a and b ports, as shown in Figure 4.1.

The next example is still rather trivial, but it highlights a few more features of Ptalon:

```
Link is {
    actorparameter A;
    inport linkIn;
    outort linkOut;
```

[1] Actor is simply the Ptolemy II word for component.

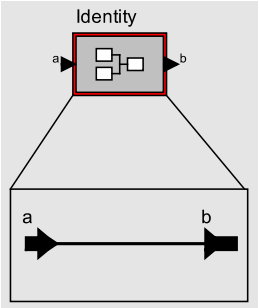

Figure 4.1: A Ptolemy model equivalent to the Identity component in Ptalon.

```
    relation r;
    A(input := linkIn, output := r);
    A(input := r, output := linkOut);
}
```

The statement `actorparameter A;` creates an actor parameter that can be assigned an actor as a value. Such parameters are what make Ptalon a higher-order composition language. The statement `relation r;` creates a relation. Like a port, a relation in Ptolemy is a connection point, but unlike a port, this connection is not exposed at the interface of the Link component. Instead it is an internal connection point that may be connected to only on the *inside* of the composite actor. The last two lines create two instances of the actor assigned to parameter `A`. If this actor has ports named input and output, the input of the first instance will be connected to the linkIn input of the Link component, and the output of the second instance will be connected to the linkOut output of the Link component. The output of the first instance will feed to the input of the second instance, passing through the relation `r`. Note that if the actor that gets passed to `A` as a parameter does not have a port

named input and output, they will be created. This can be useful, because actor code can be written to test for the presence of ports in Ptolemy.

Note that the instances of actor assigned to `A` require `A` to be assigned a value before they are created. Ptalon uses a partial evaluation strategy, which basically boils down to "Create what you can based on what parameters are known." The two actor instances may only be created when `A` is assigned a value, and this is precisely when they are created. After they are created, the Link actor corresponds to the Ptolemy model of Figure 4.2.

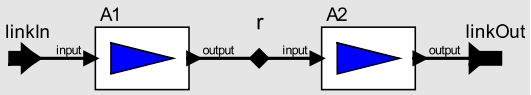

Figure 4.2: The model corresponding to the Link component in Ptalon after the actor parameter A is assigned an actor.

Now if we wanted to compose the two instance of `A` in parallel instead of in series, we can use the following Ptalon code:

```
attachDanglingPorts;

Combine is {
    actorparameter A;
    A();
    A();
}
```

The first line tells the Ptalon interpreter that any unconnected ports of an internal actor are brought to the outside. The last two lines instantiate two instances of the actor `A`. If `A` has a port named input, then Combine will have ports named `A1_input`

and `A2_input`. The Ptolemy model corresponding to the Combine code after an actor is assigned to the parameter `A` is shown in Figure 4.3.

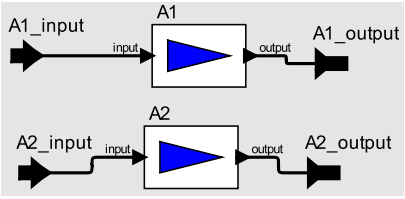

Figure 4.3: A Ptolemy model equivalent to the Combine component in Ptalon.

4.2 Recursion and Iteration

The last sections showed the features that make Ptalon a composition language in the ordinary sense. In this section, I show what recursion and iteration look like in Ptalon. Consider first a variant of `Combine` which combines `n` instances of an actor named `repeater`. I call this new higher-order component `Parallel`:

```
Parallel is {
    actorparameter repeater;
    parameter n;
    if [[n <= 1]] {
        repeater();
    } else {
        repeater();
        Parallel( repeater := repeater(), n := [[n - 1]] );
    }
}
```

Just as a notional note, the double brackets `[[]]` are used whenever an expression is to be evaluated in the Ptolemy Expression Language. Anything in between the

double brackets will get passed to the Ptolemy expression evaluator by the Ptalon interpreter, and Ptalon will simply maintain a symbol table for the expression evaluator to reference for values.

The parameter `n` is a non-higher-order parameter. It may be assigned any "token" in the Ptolemy type system as a value. Setting the parameter `n` to some number greater than 1, creates two components; one is the `repeater` component, and the other is another `parallel` component with the same `repeater` component and `n` set to `n-1`. By the Ptalon partial evaluation convention, the conditional block will not be tested and entered until after `n` is assigned a value. When `n` is assigned `3` as its value, and `repeater` is assigned an actor with ports named `input` and `output`, this corresponds to the Ptolemy model of Figure 4.4.

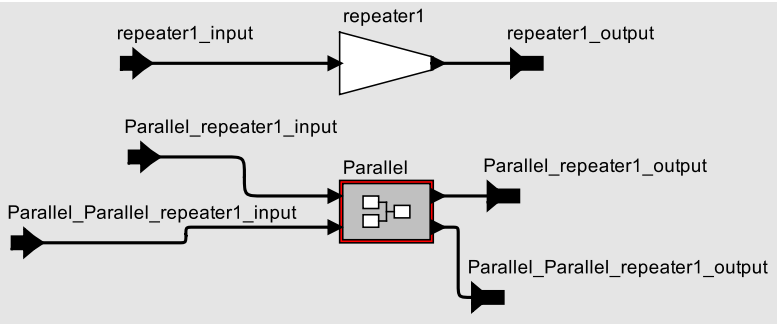

Figure 4.4: The Ptolemy model corresponding to the Parallel component in Ptalon when $n := 3$.

I can use iteration in a similar way. Consider the component:

```
IterativeParallel is {
    actorparameter repeater;
    parameter n;
    for i initially [[ 1 ]] [[ i <= n ]] {
```

```
        repeater();
    } next [[ i + 1 ]]
}
```

It has the same basic effect, but this time, it iterates over the for-loop `n` instantiating a `repeater` component each time. I show the corresponding Ptolemy model with the same parameters in Figure 4.5.

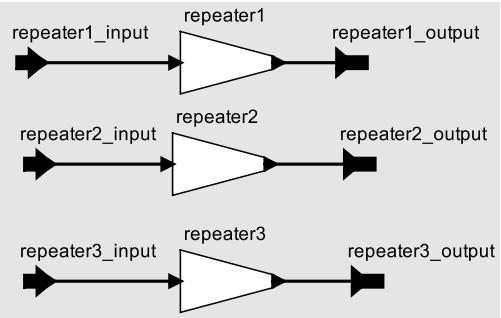

Figure 4.5: The Ptolemy model corresponding to an IterativeParallel component in Ptalon with $n := 3$.

4.3 Linear Systems–A Practical Example

The most basic and well-understood systems in control theory are linear. Callier and Desoer [9] provide a thorough introduction to such systems. A system has m inputs, described by a function $u : \mathbb{R} \to \mathbb{R}^m$, r outputs, described by a function $y : \mathbb{R} \to \mathbb{R}^r$, and n internal states, described by a function $x : \mathbb{R} \to \mathbb{R}^n$. The behavior

for a linear system is described by the equations:

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{aligned}$$

where A, B, C, and D are real matrices of appropriate dimension.

I created a higher-order component to model such components in Ptalon, based on a concept by Jie Liu. Its parameters are the parameters are the A, B, C, and D matrices of the system, as well as the initial states of the system. Setting these parameters creates the linear system using *Scale* components, to perform scalar multiplications, *Add* components, to perform scalar addition, and *Integrator* components to perform numerical integration. The generated model for these particular parameters is given in Figure 4.6:

$$A = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{4.1}$$

$$B = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{4.2}$$

$$C = \begin{pmatrix} 1 & 0 \end{pmatrix} \tag{4.3}$$

$$D = 0 \tag{4.4}$$

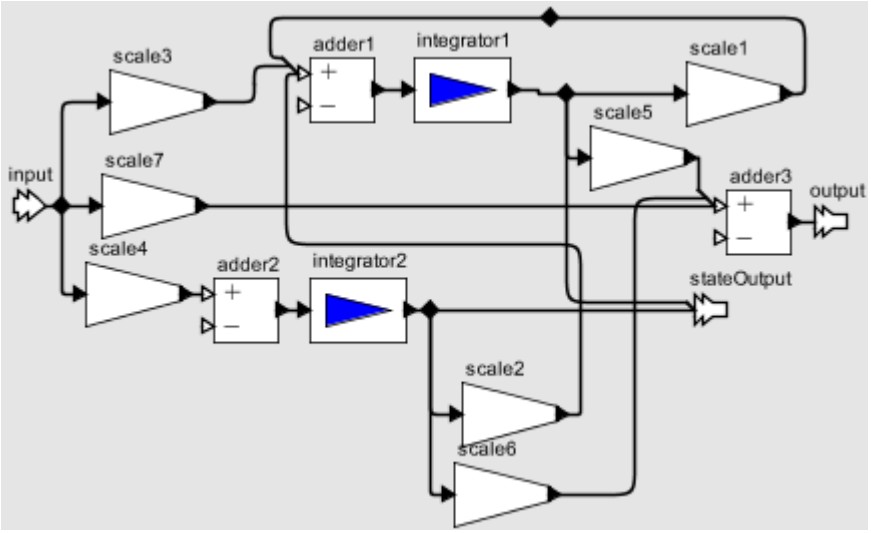

Figure 4.6: A Ptolemy model corresponding to the LinearStateSpace component in Ptalon with the parameters of Equations 4.1 through 4.4.

4.3.1 Ptalon Code

Below is the Ptalon code for the LinearStateSpace component. It uses a few advanced features of Ptalon I left out above. Most of these are explained in the code documentation, but one point that deserves explanation are *multiports*. A multiport is a single port than can act as several ports. Making three connections to the outside of a multiport is like creating three ports and connecting to them all. Here is the code:

```
/*
 * Linear state space model in the CT domain.  This is the Ptalon version
 * of a model originally written in Java code.
 *
 * The State-Space model implements a system whose behavior is defined by:
 *
 * dx/dt = Ax + Bu
 * y = Cx + Du
 * x(0) = x0
```

```
 *
 * where x is the state vector, u is the input vector, and y is the output
 * vector. The matrix coefficients must have the following characteristics:
 *
 * A must be an n-by-n matrix, where n is the number of states.
 * B must be an n-by-m matrix, where m is the number of inputs.
 * C must be an r-by-n matrix, where r is the number of outputs.
 * D must be an r-by-m matrix.
 *
 * The actor accepts m inputs and generates r outputs
 * through a multi-input port and a multi-output port. The widths of the
 * ports must match the number of rows and columns in corresponding
 * matrices, otherwise, an exception will be thrown.
 *
 *
 * @author Adam Cataldo (borrowed example from Jie Liu)
 */

LinearStateSpace is {
    /*
     * These lines create the ports for the actor, which are
     * all multiports.
     */
    inport[] input;
    outport[] output;
    outport[] stateOutput;

    /*
     * These lines create the parameters for the actor.
     */
    parameter A;
    parameter B;
    parameter C;
    parameter D;
    parameter initialStates;

    /*
     * These lines define some actor "constants" to be
     * used later.
     */
    actor integrator = ptolemy.domains.ct.lib.Integrator;
    actor adder = ptolemy.actor.lib.AddSubtract;
```

```
actor scale = ptolemy.actor.lib.Scale;

/*
 * This is a for loop in Ptalon.  All evaluation
 * of normal data is done by the Ptolemy expression
 * language, which we use in a for loop. Anything in
 * the denotation brackets [[ ... ]] will be parsed
 * in the expression language, not Ptalon, but the
 * results of evaluation will be used by Ptalon to
 * populate the actor.
 *
 * A for loop structure has form
 * for variable initially [[ initialValueExpression ]]
 *       [[ loopConditionExpression ]] {
 *    ...
 *    loopBody
 *    ...
 * } next [[ valueUpdateExpression ]]
 *
 * The variable can take on any data type the expression
 * language supports, although it will often be an integer.
 */
for a initially [[ 0 ]] [[ a < A.getRowCount ]] {

    /*
     * These lines create relations whose names
     * depend on the value of the variable a. They
     * will take on names like state0, state1, etc.
     * and stateAdderOut0, stateAdderOut1, etc.
     */
    relation state[[a]];
    relation stateAdderOut[[a]];

    /*
     * This creates an instance of the integrator.
     * It assigns the value of initialStates(0, a)
     * to the intialState parameter of the integrator.
     * It connects the input port of the integrator to
     * the relation stateAdderOut[[a]], where a is the for
     * loop variable.
     */
    integrator(initialState := [[initialStates(0, a)]],
```

```
        input := stateAdderOut[[a]], output := state[[a]] );

    /*
     * This creates a port reference.  The port
     * reference will refer to the plus input of
     * the adder.  Any other components that
     * later link a port to this port reference
     * will connect to the plus input of the
     * adder.  The rule for port references is
     * that they refer to the first port that
     * is linked to them through an equation
     * port := reference
     */
    port reference stateAdderIn[[a]];
    adder(plus := stateAdderIn[[a]],
        output := stateAdderOut[[a]] );

    /*
     * The "this" reference refers to the main PtalonActor
     * we create.  We use it to connect internal ports
     * and relations.  In this case, we connect the stateOutput
     * multiport to the relations state0, state1, etc.
     */
    this(stateOutput := state[[a]] );
} next [[ a + 1 ]]

/*
 * These nested for loops are used to create the scale actors
 * in the feedback loops corresponding to the input matrices.
 */
for a initially [[ 0 ]] [[ a < A.getRowCount ]] {
    for b initially [[ 0 ]] [[ b < A.getRowCount ]] {
        scale(factor := [[ A(a, b) ]], input := state[[b]],
            output := stateAdderIn[[a]] );
    } next [[ b + 1 ]]
} next [[ a + 1 ]]

/*
 * These nested for loops are used to create the scale actors
 * on the input side of the system.
 */
for b initially [[ 0 ]] [[ b < B.getColumnCount ]] {
```

```
        relation scaleIn[[b]];
        this(input := scaleIn[[b]] );
        for a initially [[ 0 ]] [[ a < A.getRowCount ]] {
            scale(factor := [[ B(a, b) ]], input := scaleIn[[b]],
                output := stateAdderIn[[a]] );
        } next [[ a + 1 ]]
    } next [[ b + 1 ]]

    /*
     * These nested for loops are used to create the adder actors
     * and scale actors on the output side of the system.
     */
    for c initially [[ 0 ]] [[ c < C.getRowCount ]] {
        port reference outputAdderIn[[c]];
        adder(plus := outputAdderIn[[c]], output := output);
        for a initially [[ 0 ]] [[ a < A.getRowCount ]] {
            scale(factor := [[ C(c, a) ]], input := state[[a]],
                output := outputAdderIn[[c]] );
        } next [[ a + 1 ]]
    } next [[ c + 1 ]]

    /*
     * These nested for loops are used to create the
     * and scale actors for the direct feedthrough subsystem.
     */
    for c initially [[ 0 ]] [[ c < C.getRowCount ]] {
        for b initially [[ 0 ]] [[ b < B.getColumnCount ]] {
            scale(factor := [[ D(c, b) ]], input := scaleIn[[b]],
                output := outputAdderIn[[c]] );
        } next [[ b + 1 ]]
    } next [[ c + 1 ]]

}
```

Chapter 5

Conclusions

The purpose of this dissertation is to show the power of higher-order composition languages in system design. In order to formalize this, I develop an abstract syntax for composition languages and two calculi. The first calculus serves as a framework for composition languages without higher-order components. The second is a framework for higher-order composition languages. I prove there exist classes of systems whose specification in a higher-order composition language is drastically more succinct than it could ever be in a non-higher-order composition language.

To justify the calculus, I use it as a semantic domain for a simple higher-order composition language. I use it to reason about higher-order components in this more practical language and use n-level clock distribution networks as a class of systems whose description must grow exponentially with n when higher-order components are forbidden, but whose description grows linearly with n when higher-order components

are used.

As a prototype for higher-order composition language, I developed the Ptalon programming language for Ptolemy II. I explain how components can be built in Ptalon, and I give several examples of models built as a higher-order components in this language. These examples span several domains in system design: control theory, signal processing, and distributed systems.

Unlike functional languages, where higher-order functions are infused with a program's execution semantics, the ability to provide scalable higher-order mechanism is completely separated from execution semantics in higher-order composition languages. As a design technique, higher-order composition languages can be used to provide extremely scalable system descriptions.

Bibliography

[1] Ákos Lédeczi, Árpád Bakay, Miklós Maróti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, 2001.

[2] Arvind. Bluespec: A language for hardware design, simulation, synthesis and verification. Invited talk. *MEMOCODE*, page 249, 2003.

[3] John Aynsley and David Long. Draft standard SystemC language reference manual. Technical report, Open SystemC Initiative, 2005.

[4] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *Computer*, 36(4):45–52, 2003.

[5] Henk 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.

[6] Henk Barendregt and Erik 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.

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

[8] Christopher Brooks, Edward A. Lee, Xiojun Liu, Stephen Neuendorffer, Yang Zhao, and Haiyang 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.

[9] Frank M. Callier and Charles A. Desoer. *Linear system theory*. Springer-Verlag, London, UK, 1991.

[10] Adam Cataldo, Elaine Cheong, Thomas Huining Feng, Edward A. Lee, and Andrew Mihal. A formalism for higher-order composition languages that satisfies the Church-Rosser property. Technical Report UCB/EECS-2006-48, EECS Department, University of California, Berkeley, May 9 2006.

[11] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

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

[13] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 493–501, New York, NY, USA, 1993. ACM Press.

[14] Byron Cook, John Launchbury, and John Matthews. Specifying superscalar microprocessors in hawk. In *Formal Techniques for Hardware and Hardware-like Systems*, Marstrand, Sweden, 1998.

[15] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[16] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press, 1990.

[17] James B. Dabney and Thomas L. Harman. *Mastering SIMULINK*. Prentice Hall Professional Technical Reference, 2003.

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

[19] Doron Drusinsky and David Harel. On the power of bounded concurrency i: finite automata. *J. ACM*, 41(3):517–539, 1994.

[20] Stephen A. Edwards and Olivier 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.

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

[22] Johan Eker, Jorn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, Stephen Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Appeared in: 19th ACM Symposium on Operating Systems Principles*, Lake George, NY, October 2003.

[24] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, 2003.

[25] Roger Hindley and Jonathan Seldin. *Introduction to Combinators and λ-Calculus*. Students Texts Nr. 1. London Mathematical Society, 1986.

[26] James Hoe. *Operation-Centric Hardware Description and Synthesis*. PhD thesis, MIT, June 2000.

[27] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.

[28] Paul Hudak and Joseph H. Fasel. A gentle introduction to Haskell. *SIGPLAN Not.*, 27(5):1–52, 1992.

[29] Cristopher Hylands, Edward A. Lee, Jie Liu, Xiaojun Liu, Yuhong Xiong, Yang Zhao, and Haiyang Zheng. Overview of the ptolemy project. Technical Report UCB/ERL M03/25, EECS Department, University of California, Berkeley, 2003.

[30] Ethan K. Jackson and Janos Sztipanovits. Towards a formal foundation for domain specific modeling languages. In *EMSOFT*, pages 53–62, Seoul, South Korea, October 2006. ACM.

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

[32] Gary W. Johnson and Richard Jennings. *LabVIEW Graphical Programming*. McGraw-Hill Professional, 2001.

[33] Sung-Mo (Steve) Kang and Yusuf Leblebici. *CMOS Digital Integrated Circuits: Analysis and Design*. William C Brown Publishing, 2nd edition, 1998.

[34] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonalization of concerns and platform-based design. *IEEE trans on Computer-Aided Design*, 19(12), December 2000.

[35] Mary Sheeran Koen Claessen. A lava tutorial. April 2000.

[36] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Y Thomason IV, Greg G Nordstrom, and Peter Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing*, Budapest, Hungary, May 2001.

[37] Edward 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.

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

[39] Edward A. Lee and Pravin Varaiya. *Structure and Interpretaion of Signals and Systems*. Addison Wesley, 2003.

[40] Nenand Medvidovic and Richard N. Taylor. A classification and comparison

framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[41] Albert R. Meyer and Michael J. Fischer. Economy of description by automata, grammars, and formal systems. In *FOCS*, pages 188–191, 1971.

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

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

[44] Robin Milner. *Logic and Algebra of Specification*, chapter The Polyadic π-Calculus: A Tutorial. Springer-Verlag, 1993.

[45] Alan V. Oppenheim, Ronald W. Schafer, and John R. Buck. *Discrete-time signal processing (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[46] Alan V. Oppenheim, Alan S. Willsky, and Syed Hamid Nawab. *Signals and systems*. Prentice-Hall signal processing series. Second edition, 1997.

[47] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. In Marvin V. Zelkowitz, editor, *Advances in Computers*, volume 46, pages 329–400. Academic Press, 1998.

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

[49] Hideki John Reekie. *Realtime Signal Processing - Dataflow, Visual, and Functional Programming*. PhD thesis, University of Technology at Sydney, 1995.

[50] IEEE Computer Society. IEEE Std 1364-2001, September 2001.

[51] Janos Sztipanovits and Gabor Karsai. Model-integrated computing. *Computer*, 30(4):110–111, 1997.

[52] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, 1993.