# CHOCOLATE: Calculi of Higher Order COmmunication and LAmbda TErms (Preliminary Report)

Bard Bloom*
Cornell University, Ithaca, NY
bard@cs.cornell.edu

October 29, 1993

## Abstract

We propose a general definition of higher-order process calculi, generalizing CHOCS [Tho89] and related calculi, and investigate its basic properties. We give sufficient conditions under which a calculus is finitely-branching and effective. We show that a suitable notion of higher-order bisimulation is a congruence for a subclass of higher-order calculi. We illustrate our definitions with a sample calculus strictly stronger than CHOCS.

## 1 Introduction

Higher-order process calculi provide have captured considerable interest recently. These are calculi for concurrency in which the basic elements of the system, processes or channels, are presented as first-class objects. As in sequential programming, higher-order features grant programmers a wide variety of programming methods.

For example, in first-order calculi (CCS, CSP, ACP, and so forth), servers must use fixed communication channels. Requests come to the server on channel $a$, replies return on channel $b$, and woe betide the system if anyone else uses these channels. Clients of the server will have to have some kind of mutual exclusion or other access control protocol, requiring some careful programming and introducing more ways to make errors: a bad thing in a system intended for specifications.

*Supported by NSF grants CCR-9003441 and CCR-9223183.



POPL 94- 1/94, Portland Oregon,USA



Higher-order calculi such as CHOCS [Tho89] and the related calculus of [Hen93], and the $\pi$-calculus [MPW92] address these issues. In these calculi, processes and channels may be communicated along channels. Thus, in CHOCS, a client of a server could send the server a small process capable of sending messages back to the client, thereby reducing and encapsulating the possible misuse of the server. In the $\pi$-calculus, the interface is even cleaner: the client sends the server a channel to use for the reply.

There is a distinction between two kinds of higher-order calculi. Some calculi, CHOCS and its relatives, allow transmission of *processes* along channels. This is extremely powerful, and the implications of this programming paradigm remain to be fully explored. The other school, the $\pi$-calculus and its relatives, simply allow transmission of *channels* (or channel names) along channels. Programming in this setting is somewhat harder (*e.g.*, the implementation of the $\lambda$-calculus in the $\pi$-calculus is rather more subtle than in CHOCS), and the classical operational semantics is considerably trickier. Implementation of $\pi$-calculus-like systems is much easier; *e.g.*, the communications fragment of Concurrent ML [Rep91, Rep89] resembles the $\pi$-calculus in flavor.

In this study, we make preliminary investigations of the general theory of CHOCS-like languages. (The $\pi$-calculus is rather more delicate, and will be the subject of later studies.) The operations provided with CHOCS and its relatives are the basic concurrent operations: *e.g.*, a CCS-like parallel composition $p|q$ which allows $p$ and $q$ to either execute independently or communicate.

One useful operation not definable in CHOCS is *broadcast*, $\Downarrow$. This associative, commutative binary operation has the property that, if one combines several processes:

$$p \Downarrow q \Downarrow r \Downarrow s$$

if one process, say $p$, sends a message $m$ on channel

$a$, then all of $q$, $r$, and $s$ that are currently capable of reading from channel $a$ receive $m$. Unlike anything programmable with simple parallel composition, this guarantees that *all* possible interested receivers get the message, simultaneously. Broadcast in various forms is an essential programming operation in many distributed systems [BCG91, BC90].

Another operation which cannot even be simulated easily in CHOCS is the LOTOS operation of *disabling*, [>. The process $p$ [> $q$ behaves like $p$, until such time as $q$ takes its first step; thereafter, $p$ is killed and $q$ executes. This is vital in the higher-order setting: we may receive a process, start to execute it, discover that it is misbehaving, and wish to kill it before it can do any further damage. The process $a?x.((x\backslash\{b\})\ [>\ (b?y.\mathbf{0}))$ receives a process $p$ along $a$, and runs it — but provides an "off" switch, as a signal along channel $b$ will cancel it.

Any number of other operations have been proposed and used in first-order process calculi. There are *synchronous products* which run processes in lock-step parallel. There are *polling operations*, which allow one process to ask if another is ready to communicate without actually performing the communication. Some operations are simply for programming convenience: sequential composition $p;q$ and `while` loops are useful programming constructs which can be simulated to some extent in most process calculi, but might reasonably be made part of the language.

## 1.1 Developing a Metatheory

Rather than rebuilding the basic theory whenever a new operation is needed, we investigate the metatheory of CHOCS-like process calculi. In particular, we define a type system and form of rules, called the *chocolate* rules in honor of their derivation from CHOCS. CHOCS and Hennessy's calculus are chocolate, mutatis mutandis, as are their extensions by all of the operations mentioned above and much more.

Chocolate languages have two kinds of computation. The primary form is process-algebraic: processes produce signals (carrying other processes as data) and evolve into other processes, much as in CCS and CHOCS. The other form of computation is reduction in a typed $\lambda$-calculus.[1] For example, a term that is prepared to receive a value is written as a function taking values as arguments. Giving it a value corresponds to function application. This application is evaluated, eventually yielding a process capable of engaging in further communication. The evaluation part of computation is hidden inside the two or three relevant rules in CHOCS; it is made explicit (though instantaneous) in chocolate.

Some basic properties hold for all chocolate languages: in particular, the evaluation part of computation is strongly normalizing (which is no great surprise, as the evaluation part of the calculus is little more than simply typed $\lambda$-calculus.) We use this fact to give sufficent conditions on chocolate languages guaranteeing some important properties; *e.g.*, explaining when the transition relation is computable and finitely branching. We illustrate with NESTLÉ, an extension of CHOCS including broadcast and channel creation and transmission.

We then investigate notions of bisimulation. The most natural and useful definition of bisimulation at all types is dangerous; technically, it is not a monotone functional and hence does not necessarily have a greatest fixed point. The theory may work out appropriately, but this remains to be shown.

Instead, we provide a generalization of Thomsen's notion of higher-order bisimulation. Thomsen's notion is *not* an instance of the natural notion, as we explain in Section 4. Nonetheless, in a restricted second-order case, Thomsen's notion can be generalized to the concept of *white chocolate bisimulation*.[2] We show that white chocolate bisimulation is a congruence for a respectable class of operations, including those of CHOCS and Hennessy's calculus. This suffices for most of the operations we used to motivate this study. Unfortunately, it fails for the most interesting example, the channel transmission part of the calculus NESTLÉ.

## 1.2 Types and Channels

A fully-fledged higher-order process calculus should allow the kinds of programming paradigms that make higher-order sequential computation powerful: encapsulating and transmitting programming constructs, such as transmission and reception operations. That is, we should be able to transmit *channels*, rather in the style of the $\pi$-calculus. Keeping track of types will prove informative.

Let $\mathbf{P}$ be the type of processes. A receiver is a function that takes an input (of type $\mathbf{P}$) and then acts like a process. It is thus an *abstraction*, of type

$$\mathbf{Ab} = \mathbf{P} \to \mathbf{P}$$

A sender produces a datum (of type $\mathbf{P}$) as output, and then continues running its main thread of communication. That is, it simply is a pair of processes $\langle d, p\rangle$: $d$ to be used as data, $p$ for computation. It is thus a *concretion* [Mil91], of type

$$\mathbf{Co} = \mathbf{P} \times \mathbf{P}$$

[1] Typing is not essential to the definition of computation. It *is* essential to many finiteness and computability theorems.

[2] It's good, but it's not real chocolate.

Most higher-order process algebras have send and receive operations, written something like $a!v.p$ and $a?x.p$. We follow [Mil91] in considering $a!$ and $a?$ to be the operations. $a!$ takes a value and a process, and returns a process. $a?$ takes a function from values to processes, returning a process. Thus, we have

$$\begin{array}{rcl} a! & : \mathbf{Snd} & = \mathbf{Co} \to \mathbf{P} \\ a? & : \mathbf{Rcv} & = \mathbf{Ab} \to \mathbf{P} \end{array}$$

Given $r : \mathbf{Rcv}$, we read a value from $r$ and call it $v$ by a construct of the form $r(\lambda v.\Box)$. Given $s : \mathbf{Snd}$, we transmit $n$ along it by $s(\langle n, \Box\rangle)$. In both cases, the $\Box$ is the continuation: the process to be executed after the communication.

A communication channel consists of a way to send, and a way to receive: a pair $\langle a!, a?\rangle$, of matching sending and receiving capabilities. They thus have type:

$$\mathbf{Chan} = \mathbf{Snd} \times \mathbf{Rcv}$$

If, in our process algebra, we are able to communicate values of type **Chan**, then we can do much $\pi$-calculus-like programming straightforwardly. For example, consider a server: a process which repeatedly accepts a **Chan**, *viz.* a pair $\langle s, r\rangle : \mathbf{Snd} \times \mathbf{Rcv}$, reads a value $v$ from the channel, performs some computation $F$, and sends its result $F(v)$ back along the channel. The server should be reentrant: once it has gotten a channel, it can fork off a process $r\big(\lambda v.s\langle F(v), \mathbf{0}\rangle\big)$, abbreviated as $rv\,.\,s(F(v))\,.\,\mathbf{0}$, to handle the request along that channel; the server itself would wait for more requests. Each request can use a different channel.

$$S = a?\langle s, r\rangle\,.\,\big(S|(rv\,.\,s(F(v))\,.\,\mathbf{0})\big) \qquad (1)$$

Or, without abbreviations:

$$S = a?\Big(\lambda h.S|\big((\pi_1 h)(\lambda v.(\pi_0 h)\langle F(v), \mathbf{0}\rangle)\big)\Big)$$

A first draft of a client (sending first the channel $b$, as the value $\langle b!, b?\rangle$; then sending the value 3 coded in some unspecified way, and sending the answer along the channel *print*) would look like:

$$C_0 \;=\; a!\langle b!, b?\rangle\,.\,b!3\,.\,b?x\,.\,print!x\,.\,\mathbf{0}$$

To make full use of this, we should have a process which generates new channels on request. Suppose that there are actions $c_0$, $c_1$, .... We'd like a process with roughly the following behavior, repeatedly offering new channels as values on some fixed channel $c$:

$$NC(n) \;=\; c!\langle c_n!, c_n?\rangle\,.\,NC(n+1)$$

(A somewhat more powerful version of the new-channel process will be provided as a primitive.) A better client would first get a channel from $NC$, and use that channel to communicate with the server.

$$C_1 \;=\; c?\langle s, r\rangle\,.\,a!\langle s, r\rangle\,.\,s3\,.\,rx\,.\,print!x\,.\,\mathbf{0} \qquad (2)$$

The whole system will simply consist of the server, the client, and the channel server in parallel:

$$System = (S|C_1|NC(1))\backslash\{a, b, c, c_1, c_2, \ldots\}.$$

Note that the $c!$ and $c?$ used in $NC$ and $C_1$ are higher-order communication operations, as they are being used to communicate values of type **Chan**.

$$\begin{array}{rcl} c? & : & (\mathbf{Chan} \to \mathbf{P}) \to \mathbf{P} \\ & = & \left(\left(\begin{array}{l}(\mathbf{P} \times \mathbf{P} \to \mathbf{P}) \\ \times \\ ((\mathbf{P} \to \mathbf{P}) \to \mathbf{P})\end{array}\right) \to \mathbf{P}\right) \to \mathbf{P} \end{array}$$

This development of channel transmission was quite natural and straightforward. It is somewhat surprising that it requires constants of such high types. We are now beyond the scope of existing process algebras, even higher-order ones, and must check to see that the foundations are solid.

### 1.2.1 Typed Transition Relations

Hennessy's calculus has two actions $a?$ and $a!$ for each channel, and rules

$$a?\lambda x.p \xrightarrow{a?} \lambda x.p \qquad\qquad a!\langle q, p\rangle \xrightarrow{a!} \langle q, p\rangle$$

We have transitions $p \xrightarrow{a,\,\xi} q$, where $p : \mathbf{P}$, $a$ is an action, $\xi$ is a type, and $q$ has type $\xi$. (The definitions are sketched in Section 2.4.) The type information renders the ? and ! redundant. In our system, those transitions are

$$a^{\mathbf{Ab}}\lambda x.p \xrightarrow{a,\,\mathbf{Ab}} \lambda x.p \qquad\qquad a^{\mathbf{Co}}\langle q, p\rangle \xrightarrow{a,\,\mathbf{Co}} \langle q, p\rangle$$

Type information distinguishs between sending and recieving, so we use the same operation and action symbols.

This notion of transition is sensible in practice. It has the somewhat disturbing property that it doesn't respect types in any sense: a process – that is, a closed term of type $\mathbf{P}$ – can take $a$-transitions to terms of two different types. This presents no difficulties, though it may seem odd.

# 2 Calculi

We generalize the notion of a *GSOS language* [BIM88, Blo89]. GSOS languages are a first-order generalization of the rules used to define CCS.[3] We add a type system and higher-typed terms.

[3] The acronym GSOS currently stands for "Grand Structured Operational Semantics".

## 2.1 Types and Terms

The type system is straight from the simply typed $\lambda$-calculus with products and one base type, which we call $\lambda^{\rightarrow,\times}$. Types $\xi$ have the form:

$$\xi ::== \mathbf{P} \mid \xi \rightarrow \xi \mid \xi \times \xi$$

**Definition 2.1** *A* Higher Order GSOS Signature *is a set* Act *of* actions, *a set of* operation symbols, *and a vector of $k_f$ types $\langle \xi_{f1}, \ldots, \xi_{fk_f} \rangle$ for each operation symbol $f$, signifying that $f$ takes arguments of types $\xi_{f1}, \ldots, \xi_{fk_f}$ and returns a value of type $\mathbf{P}$.*

We assume infinitely many variables $x^\xi$ of each type. We define terms and their types simultaneously:

$$\frac{}{x^\xi : \xi} \qquad \frac{t_i : \xi_{fi}, 1 \le i \le k_f}{f(\vec{t}) : \mathbf{P}}$$

$$\frac{t : \xi_0 \times \xi_1}{\pi_i(t) : \xi_i} \qquad \frac{t_i : \xi_i, i \in \{0,1\}}{\langle t_0, t_1 \rangle : \xi_0 \times \xi_1}$$

$$\frac{t : \xi_0 \rightarrow \xi_1, t' : \xi_0}{tt' : \xi_1} \qquad \frac{t : \xi_1}{\lambda x^{\xi_0}.t : \xi_0 \rightarrow \xi_1}$$

We use a number of fairly obvious abbreviations to make processes easier to read. As we are combining process algebra and $\lambda^{\rightarrow,\times}$, we have in effect two computation systems, plus a rule connecting them.

## 2.2 The languages NESTLÉ and NESTLÉ/2

Based on the considerations above, we define two higher-order process calculi, NESTLÉ and its second-order subset NESTLÉ/2 . NESTLÉ includes all the operations of CHOCS, plus full higher-order communication, broadcast, disabling, and channel creation; it illustrates all the features of the rules given in our study. Our action alphabet Act is infinite; $a, b, c$ range over it. It includes an infinite, coinfinite subset $\{c_0, c_1, \ldots\}$, and a distinguished symbol $\tau$. The operation symbols are:

| | |
|---|---|
| $\mathbf{0} : \langle\rangle$ | Null process |
| $a^\xi : \langle \xi \rangle$ | Prefixing at all types, for each action $a$ |
| $+ : \langle \mathbf{P}, \mathbf{P} \rangle$ | Nondeterministic choice |
| $\mid : \langle \mathbf{P}, \mathbf{P} \rangle$ | Parallel Composition |
| $[> : \langle \mathbf{P}, \mathbf{P} \rangle$ | Disabling |
| $\backslash S : \langle \mathbf{P} \rangle$ | Restriction: forbid actions in $S \subseteq$ Act |
| $[R] : \langle \mathbf{P} \rangle$ | Renaming $a$'s to $R(a)$'s, |
| : | where $R :$ Act $\rightarrow$ Act |
| $\Psi_C : \langle \mathbf{P}, \mathbf{P} \rangle$ | Broadcasting on channels in $C \subseteq$ Act |
| $\lceil \cdot \rceil S : \langle \mathbf{P} \rangle$ | Delimit broadcasting |
| $\mathsf{NC}_n : \langle\rangle$ | New channel creator: |
| : | absent from NESTLÉ/2 |

We write operations in infix in fairly obvious ways. We use the abbreviations $a!p.q = a\langle p, q \rangle$ and $a?x.p = a(\lambda x.p)$. We have one rule for each choice of $a$, $\xi$, $C$, $S$, and $R$. The rules for prefixing and choice are quite standard:

$$\frac{}{a^\xi x \xrightarrow{a,\xi} x} \qquad \frac{x \xrightarrow{a,\xi} y}{x + x' \xrightarrow{a,\xi} y} \qquad \frac{x \xrightarrow{a,\xi} y}{x' + x \xrightarrow{a,\xi} y}$$

For parallel composition, we define $y|_\xi x'$ (where $y : \xi$ and $x' : \mathbf{P}$) to be an abbreviation for the following term, which has the same communication effect as $y$ does, but has $x'$ still running in parallel with it. That is, if $y$ wants to receive a value (*i.e.*, has type $\xi_0 \rightarrow \xi_1$), then $y|_{\xi_0 \rightarrow \xi_1} x'$ will receive a value and route it to $y$. Similarly, if $y$ wants to transmit a value (viz, has type $\xi_0 \times \xi_1$), $y|_{\xi_0 \times \xi_1} x'$ will transmit the same value $\pi_0 y$, and continue running the rest of $y$ in parallel with $x'$. This is the natural generalization of [Hen93].

$$\begin{aligned} y|_{\mathbf{P}} x' &= y|x' \\ y|_{\xi_0 \times \xi_1} x' &= \langle (\pi_0 y),\ (\pi_1 y)|_{\xi_1} x' \rangle \\ y|_{\xi_0 \rightarrow \xi_1} x' &= \lambda z^{\xi_0}.(yz|_{\xi_1} x') \end{aligned}$$

Note that $y|_\xi x : \xi$. We omit the symmetric rules, and use informal pattern matching notation for clarity.

$$\frac{x \xrightarrow{a,\xi} y}{x|x' \xrightarrow{a,\xi} y|_\xi x'} \qquad \frac{x \xrightarrow{a,\xi\rightarrow\mathbf{P}} y, \quad x' \xrightarrow{a,\xi\times\mathbf{P}} \langle y'_0, y'_1 \rangle}{x|x' \xrightarrow{\tau,\mathbf{P}} (yy'_0)|y'_1}$$

For example,

$$\begin{aligned} p_0 &= (a?x.p)|q \xrightarrow{a,\mathbf{P}\rightarrow\mathbf{P}} \lambda z.(pz|q) \\ p_1 &= (a!s.t)|u \xrightarrow{a,\mathbf{P}\times\mathbf{P}} \langle s, t|u \rangle \end{aligned}$$

then $s$ is sent to $p$ as desired in $p_0|p_1$:

$$p_0|p_1 \xrightarrow{\tau,\mathbf{P}} (ps|q)|(t|u)$$

Disabling $p$ [> $q$ runs $p$ until $q$ takes a step, then kills $p$ and runs $q$. The type extension [>$_\xi$ is defined in the same way as $|_\xi$.

$$\frac{x \xrightarrow{a,\xi} y}{x \,[> x' \xrightarrow{a,\xi} y \,[>_\xi x'} \qquad \frac{x' \xrightarrow{a,\xi} y'}{x \,[> x' \xrightarrow{a,\xi} y'}$$

Restriction and renaming must be extended to higher types in much the same way that parallel composition is. Let $t : \xi$; we define $f_\xi(t)$ as follows:

$$\begin{aligned} f_{\mathbf{P}}(t) &= t \\ f_{\xi_0 \rightarrow \xi_1}(t) &= \lambda z^{\xi_0}.f_{\xi_1}(tz) \\ f_{\xi_0 \times \xi_1}(t) &= \langle \pi_0 t, f_{\xi_1}(\pi_1 t) \rangle \end{aligned}$$

The restriction and renaming rules are fairly standard:

$$\frac{x \xrightarrow{a,\xi} y \text{ for } a \notin S}{x\backslash S \xrightarrow{a,\xi} y\backslash_\xi S} \qquad \frac{x \xrightarrow{a,\xi} y}{x[R] \xrightarrow{R(a),\xi} y[R]_\xi}$$

Broadcasting $x \Downarrow_C x'$ is similar to parallel composition. It has the parallel composition rules for $a \notin C$, plus the following rules and their symmetric versions. If $x$ broadcasts $y_0$, then $x'$ must receive it if it can; either way, $x \Downarrow_C x'$ will broadcast $y_0$.

$$\frac{x \xrightarrow{a,\xi\times\mathbf{P}} \langle y_0, y_1\rangle, \quad x' \xrightarrow{a,\xi\to\mathbf{P}} y'}{x \Downarrow_C x' \xrightarrow{a,\xi\times\mathbf{P}} \langle y_0, (y_1 \Downarrow_C y' y_0)\rangle}$$

$$\frac{x \xrightarrow{a,\xi\times\mathbf{P}} \langle y_0, y_1\rangle, \quad x' \xrightarrow{a,\xi\to\mathbf{P}} \not\to}{x \Downarrow_C x' \xrightarrow{a,\xi\times\mathbf{P}} \langle y_0, y_1 \Downarrow_C x'\rangle}$$

Note that a broadcast always emits a value $y_0$. If $p \Downarrow_C q \xrightarrow{a,\xi} r$ for some $a \in C$, then $r$ is always a pair $\langle r_0, r_1\rangle$ where $r_0$ was the value broadcast and $r_1$ the system after the broadcast occurred. Recall that if $t \xrightarrow{a,\xi} u$, then $t$ must be a process. We thus need a broadcast-delimiting operation $\lceil x\rceil C$ which strips off the value being broadcast and lets the remaining process continue.

$$\frac{x \xrightarrow{a,\xi\times\mathbf{P}} y, \quad a \in C}{\lceil x\rceil C \xrightarrow{\tau,\mathbf{P}} \pi_1 y} \qquad \frac{x \xrightarrow{a,\xi} y, \quad a \notin C}{\lceil x\rceil C \xrightarrow{a,\xi} \lceil x\rceil C, \xi}$$

$$\frac{x \xrightarrow{a,\mathbf{P}} y}{\lceil x\rceil C \xrightarrow{a,\mathbf{P}} \lceil y\rceil C}$$

So, the idiom for broadcast is

$$\lceil p \Downarrow_C q \Downarrow_C \cdots \Downarrow_C r \Downarrow_C s\rceil C$$

much as the idiom for parallel composition with point-to-point communication is

$$(p|q|\cdots|r|s)\backslash S.$$

The new channel creator $\mathsf{NC}_n$ simply offers new communication channels (on channel $c_0$). Let $\mathbf{Chan}(\xi) = (\xi \to \mathbf{P}) \times (\xi \times \mathbf{P})$.

$$\mathsf{NC}_n \xrightarrow{c_0,\,\mathbf{Chan}(\xi)\times\mathbf{P}} \left\langle \left\langle \lambda x.{c_n}^{\xi\to\mathbf{P}}(x), \lambda x.{c_n}^{\xi\times\mathbf{P}}(x)\right\rangle, \mathsf{NC}_{n+1}\right\rangle$$

Recursive definition of processes is straightforward as a consequence of our general theory: we allow constants defined by an arbitrary set of axioms. For example, the server of (1) would be defined as a new constant $S$, with a single rule

$$S \xrightarrow{a,\,\mathbf{Chan}\to\mathbf{P}} \lambda\langle s, r\rangle.(S|r(\lambda v.s\langle F(v), \mathbf{0}\rangle)).$$

NESTLÉ is a remarkably powerful calculus, allowing CHOCS-like and $\pi$-calculus-like programming. With a bit of care (having a single NC operator in parallel, and avoiding use of $c_n$ for $n > 0$), programming with dynamically-allocated channels is straightforward.

The only $\pi$-calculus operation missing from NESTLÉ is matching: $[a = b]q$ behaves like $q$ if the channel names $a$ and $b$ are the same, and is stopped otherwise. This is programmable (with some extra $\tau$-moves) in NESTLÉ.

Suppose that we have two channel values of the same type $\xi$, $\chi_1 = \langle s_1, r_1\rangle$ and $\chi_2 = \langle s_2, r_2\rangle$, produced by NC. The two are equal if a signal sent on $s_1$ is received on $r_2$. We use broadcasting to force the communication.

Let $\mathbf{0}^\xi$ be any closed term of type $\xi$. Choose channels $a$, $b_y$, and $b_n$. The following code will send a signal on $b_y$ if $\chi_1 = \chi_2$, and on $b_n$ otherwise. Let $G = \{a, c_0, c_1, \ldots\}$;

$$\begin{aligned} p_1 &= (r_2?x\,.\,b_y!\mathbf{0}\,.\,\mathbf{0}) + (a?x\,.\,b_n!\mathbf{0}\,.\,\mathbf{0}) \\ p &= \left\lceil (s_1!\mathbf{0}^\xi\,.\,a!\mathbf{0}\,.\,\mathbf{0}) \Downarrow_G p_1 \right\rceil G\backslash G \end{aligned}$$

If $\chi_1 = \chi_2$, this will evolve as follows:

$$p \xrightarrow{\tau,\mathbf{P}} \lceil a!\mathbf{0}.\mathbf{0} \Downarrow_G b_y!\mathbf{0}.\mathbf{0}\rceil G\backslash G \xrightarrow{b_y,\,\mathbf{Co}} \langle \mathbf{0}, \Box\rangle$$

where $\Box$ is $\mathbf{0}$ inside some communication delimiters. (There might be another $\tau$-move at one end or the other.) The use of $\Downarrow_G$ forces the communication between $s_1$ and $r_2$ to occur if it is possible.

On the other hand, if $\chi_1 \neq \chi_2$, then the left-hand process broadcasts into the void; we have only a single computation:

$$\begin{aligned} p &\xrightarrow{\tau,\mathbf{P}} \lceil a!\mathbf{0}.\mathbf{0} \Downarrow_G p_1\rceil G\backslash G \\ &\xrightarrow{\tau,\mathbf{P}} \lceil \mathbf{0} \Downarrow_G b_n!\mathbf{0}\,.\,\mathbf{0}\rceil G\backslash G \\ &\xrightarrow{b_n,\,\mathbf{Co}} \langle \mathbf{0}, \Box\rangle \end{aligned}$$

Thus, $p$ can tell if two channels are the same or not.

NESTLÉ/2 is NESTLÉ restricted to second-order types: no channel creation, and only tuples of processes can be transmitted. It is strictly between CHOCS and NESTLÉ in expressive power, and our stronger theory applies to it.

## 2.3 Operational Semantics

A chocolate language consists of a signature and some structured operational rules defining the behavior of the

operators of the signature. The $\lambda$-calculus fragment is standard.

**Definition 2.2** *Higher Order GSOS rules have the form*

$$\frac{x_1 \xrightarrow{a_{11},\xi_{11}} y_{11}, \quad \ldots, \quad x_1 \xrightarrow{b_{11},\xi'_{11}}, \quad \ldots,}{f(\vec{x}) \xrightarrow{c} u} \tag{3}$$

*There may be infinitely many antecedents, both positive and negative; $u$ may be any term. The variables $x_i$ and $y_{ij}$ must all be distinct, and the only variables appearing in the term $u$ are $x_i$'s and $y_{ij}$'s.*

**Definition 2.3** *A* chocolate language $\mathcal{L}$ *over a signature is a set of higher-order GSOS rules over that signature.*

The evaluation rules for $\lambda^{\rightarrow,\times}$ are standard, and not left up to the language designer.[4]

$$\frac{}{(\lambda x.t)u \Longrightarrow t[x := u]} \qquad \frac{}{\pi_i(\langle t_0, t_1\rangle) \Longrightarrow t_i}$$

$$\frac{t \Longrightarrow t'}{tu \Longrightarrow t'u} \qquad \frac{t \Longrightarrow t'}{\pi_i(t) \Longrightarrow \pi_i(t')}$$

The transition relation $\Longrightarrow$ is deterministic. By standard theories of typed $\lambda$-calculus, it is strongly normalizing; that is, there are no infinite reductions sequences $p_0 \Longrightarrow p_1 \Longrightarrow \cdots$.

Chocolate calculi all use the *Instanteous Evaluation Rule*: local computation is instantaneous, and only communication takes time. This is actually moderately realistic, in software systems at least: local processing time are generally insignificant compared to communication delays.

$$\frac{x \Longrightarrow x', \quad x' \xrightarrow{a,\xi} y}{x \xrightarrow{a,\xi} y} \tag{4}$$

In the full paper we explain why having evaluation steps take $\tau$-moves is undesirable.

For example, let $p$ be a process which receives a channel along $a$, sends some $d$ along that channel, and stops; and $q$ be a process which sends a channel $b$ along $a$, reads a value from $b$, and thereafter behaves like that value.

$$\begin{aligned} p &= a?(\lambda c.\pi_0(c)\langle d, \mathbf{0}\rangle) \\ q &= a!\langle\langle b!, b?\rangle, b?(\lambda x.x)\rangle \end{aligned}$$

[4]It would probably make good sense to use a more powerful calculus, though the finiteness results are unlikely to hold if the base calculus is not strongly normalizing.

We have transitions:

$$p \xrightarrow{\tau,\ \mathbf{Chan}\rightarrow\mathbf{P}} \lambda c.\pi_0(c)\langle d, \mathbf{0}\rangle$$

$$q \xrightarrow{a,\ \mathbf{Chan}\times\mathbf{P}} \langle\langle b!, b?\rangle, b?(\lambda x.x)\rangle$$

and hence

$$p|q \xrightarrow{a,\mathbf{P}} ((\lambda c.\pi_0(c)\langle d, \mathbf{0}\rangle)\langle b!, b?\rangle)|b?(\lambda x.x) = p'|q'$$

Only the evaluation rule applies to $p'$. We have

$$\begin{aligned} ((\lambda c.\pi_0(c)\langle d, \mathbf{0}\rangle)\langle b!, b?\rangle) &\Longrightarrow \pi_0(\langle b!, b?\rangle)\langle d, \mathbf{0}\rangle \\ &\Longrightarrow b!\langle d, \mathbf{0}\rangle \xrightarrow{b,\mathbf{Co}} \langle d, \mathbf{0}\rangle \end{aligned}$$

which matches $q'$'s transition $q' \xrightarrow{b,\mathbf{Ab}} \lambda x.x$ to give us a transition

$$p'|q' \xrightarrow{b,\mathbf{P}} \mathbf{0}|(\lambda x.x)d$$

That is, $p$ has sent $d$ to $q$, which is what we intended.

## 2.4 Definition of the Transition Relation

A transition $p \xrightarrow{a,\xi} q$ occurs in $\mathcal{L}$ iff there is a proof of this formula from the rules of $\mathcal{L}$, plus the evaluation rules and the connection rule. The formal definition in the full paper is nontrivial, especially as we must define proofs for $p \xrightarrow{a,\xi}$ as well. The following lemma shows that the definitions fit together right:

**Lemma 2.4** *Let $\mathcal{L}$ be any chocolate language, and $p$ : $\mathbf{P}$ any closed term. Then there is a $\mathcal{L}$-proof of $p \xrightarrow{a,\xi} q$ for some $q$ iff there is no $\mathcal{L}$-proof of $p \xrightarrow{a,\xi}$.*

# 3 Computability and Finite Branching

There are some fundamental concerns about this calculus. Clearly, as we have allowed arbitrary sets of operation symbols, actions, and rules, we cannot expect anything to be computable, or even countable. It is often desirable to keep the transition relation computable and finitely branching.

## 3.1 Finite Branching

It is frequently important to have some kind of finiteness and computability properties. We discuss two important finite-branching properties.

Suppose that $\Pi$ is a proof about $p$. The immediate subproofs of $\Pi$ are either about the arguments of $p$ (if $p = f(\vec{p})$), or about a term $p'$ with $p \Longrightarrow p'$. This suggests the following definition:

**Definition 3.1** $p \rhd\!\!\!\multimap q$ *iff either* $p \Longrightarrow q$, *or* $p = f(p_1, \ldots, p_n)$ *and* $q = p_i$ *for some* $i$ *such that* $p_i : \mathbf{P}$.

$\rhd\!\!\!\multimap$ is a lousy notion of reduction: it is nondeterministic, non-confluent (though finitely branching), and not respecting of meaning in any sense. The main excuse for $\rhd\!\!\!\multimap$ is the following:

**Lemma 3.2** *Let* $\Pi$ *be a proof about* $p$. *If* $\Pi$ *contains a subproof about a term* $p'$, *then* $p \overset{*}{\rhd\!\!\!\multimap} p'$.

The usual proofs for strong normalization of the simply typed $\lambda$-calculus can be adapted to show that $\rhd\!\!\!\multimap$ is strongly normalizing and finitely branching. Thus, the following function is well-defined:

$$\mathsf{dur}(p) = \max\left\{n | \exists p'.p \overset{n}{\rhd\!\!\!\multimap} p'\right\}$$

The heart of the finiteness and computability proofs are the following lemma, where $\mathsf{depth}(\Pi)$ is the depth of a proof in the obvious sense.

**Lemma 3.3** *Let* $\Pi$ *be a proof about* $p$. *Then* $\mathsf{depth}(\Pi) \leq \mathsf{dur}(p)$.

**Definition 3.4** $\mathcal{L}$ *is* image finite *if, for all processes* $p$, *actions* $a$, *and types* $\xi$, *the set* $\left\{p' | p \xrightarrow{a,\xi} p'\right\}$ *is finite.* $\mathcal{L}$ *is* finitely branching *if, for all processes* $p$, *the set* $\left\{\langle a, \xi, p'\rangle | p \xrightarrow{a,\xi} p'\right\}$ *is finite.*

Image finiteness is the weakest finite-branching property in general use. In many calculi, it is the strongest property that we want. The NESTLÉ operation $\mathsf{NC}_n$ is image-finite, but (as it offers channels of all types) is not finitely branching.

**Theorem 3.5** *Suppose that there are only a finite number of rules for* $f(\vec{x}) \xrightarrow{a,\xi}$ *for each* $f$, $a$, *and* $\xi$, *and that all rules have a finite number of positive antecedents. Then* $\mathcal{L}$ *is image finite.*

The essence of the following method to get finite branching is taken from [Ace92, Vaa93].

**Definition 3.6** *The* trigger *of a rule* $\rho$ *in the form (3) is the vector* $\langle S_1, \ldots, S_n\rangle$, *where*

$$S_i = \left\{\langle a_{ij}, \xi_{ij}\rangle | \left(x_i \xrightarrow{a_{ij},\xi_{ij}} y_{ij}\right) \in \mathsf{ante}(\rho)\right\}$$

**Definition 3.7** *A chocolate language is* finitely triggered *if, for each operation symbol* $f$ *and potential trigger* $T$, *there are only finitely many rules for* $f$ *with trigger* $T$; *and each rule has only finitely many positive antecedents.*

NESTLÉ is not finitely triggered, as there are infinitely many rules for $\mathsf{NC}_n$ with the empty tuple for a trigger. Without $\mathsf{NC}_n$ — or even with $\mathsf{NC}_n$ restricted to a finite set of types — it would be finitely triggered.

**Theorem 3.8** *If* $\mathcal{L}$ *is finitely triggered, then it is finitely branching.*

## 3.2 Computability

There are several possible definitions of computability. For this study, we choose one of the strongest: that the computation tree of a process is computable.

**Definition 3.9** *Let* $D(p) = \left\{\langle a, q\rangle | p \xrightarrow{a,\mathbf{P}} q\right\}$. $\mathcal{L}$ *is* strongly computable *iff* $D(p)$ *is computable in canonical index as a function of* $p$.

By insisting that appropriate parts of Lemma 3.8 be computable, we can give sufficient conditions for strong computability.

**Lemma 3.10** *Let* $\mathcal{L}$ *be a chocolate language, such that:*

1. *The sets of actions, operations, and rules have recursive codings, such that equality is recursive.*
2. *All rules have finite numbers of positive antecedents, and recursive sets of negative antecedents; and that the set of antecedents in canonical index and a program deciding the negative antecedents are computable from the code of rule.*
3. *For any operation symbol and finite trigger, the set of rules for that symbol and trigger is computable in canonical index.*

*Then* $\mathcal{L}$ *is strongly computable.*

These conditions are not sufficient, for stupid reasons at least: *e.g.*, a thoroughly nonrecursive language without constants would have no closed terms, and thus no processes and an easily computible (*viz.*, empty) transition relation.

# 4 Bisimulation as a Congruence: White Chocolate

There are technical obstacles to defining bisimulation for chocolate languages in general. The definition that we would like is

**Definition 4.1** *A relation between closed terms of the same type,* $\backsim$, *is a* $\forall$ bisimulation relation *if it is symmetric and whenever* $p \backsim p'$:

- *If* $p, p' : \mathbf{P}$, *then whenever* $p \xrightarrow{a,\xi} q$, *there is a* $q' \backsim q$ *such that* $p' \xrightarrow{a,\xi} q'$.

- *If* $p, p' : \xi_0 \times \xi_1$ *then* $\pi_i p \backsim \pi_i p'$ *for* $i = 0, 1$.

- *If* $p, p' : \xi_0 \to \xi_1$, *then for all* $q \backsim q'$ *of type* $\xi_0$, $pq \backsim p'q'$.

(This is a kind of late bisimulation.) We would then define $\leftrightarrows$ to be the largest $\forall$ bisimulation relation. It works for first-order processes; the first-order parts of this definition are monotone, and thus there is indeed a well-defined notion of $\leftrightarrows$. However, for higher-order processes, the universal quantifier in the $\xi_0 \to \xi_1$ case of the definition makes the definition non-monotone; the more processes of type $\xi_0$ which are related, the harder it is to be related at type $\xi_0 \to \xi_1$. It is an open question whether or not there is a maximum $\forall$ bisimulation relation.

There are several variant definitions, replacing the universal quantifier at functional type with an existential: if $p \backsim p' : \xi_0 \to \xi_1$, for every $q$ there is a $q' \backsim q$ such that $pq \backsim p'q'$. This definition is monotone at least, so there is a well-defined $\leftrightarrows$ relation, but the weaker definition complicates the analysis. For this study, we restrict attention to a much smaller class of languages, scarcely more powerful than CHOCS. Definitions specific to this weaker class are termed "white chocolate."

We take a restricted type system, the *White Chocolate Type System:*

$$\xi := \mathbf{P} | \mathbf{P}^n | \mathbf{P}^n \to \mathbf{P}$$

where $n \geq 0$. We do not bother to identify $\mathbf{P}$, $\mathbf{P}^1$, and $\mathbf{P}^0 \to \mathbf{P}$. We define the natural generalization of Thomsen's notion of higher-order bisimulation, which does not try to relate functions directly; instead, it applies them arguments in all ways and relates the resulting processes.

Let $\mathsf{nf}(q)$ be the $\Longrightarrow$-normal form of $q$.

**Definition 4.2** *A relation* $\backsim$ *between closed terms of type* $\mathbf{P}$ *is a* white bisimulation relation *iff it is symmetric, and whenever* $p \backsim p'$, *we have:*

**P:** *If* $p \xrightarrow{a,\mathbf{P}} q$, *then there exists* $q' \backsim q$ *such that* $p' \xrightarrow{a,\mathbf{P}} q'$.

**$\mathbf{P}^k$:** *If* $p \xrightarrow{a,\mathbf{P}^k} q$, *then there exists* $q'$ *such that* $p' \xrightarrow{a,\mathbf{P}} q'$, *such that* $\mathsf{nf}(q)$ *is componentwise* $\backsim$ *to* $\mathsf{nf}(q')$.

**$\mathbf{P}^k \to \mathbf{P}$:** *If* $p \xrightarrow{a,\mathbf{P}^k\to\mathbf{P}} q$, *then for every closed* $r : \mathbf{P}^k$, *there is a* $r'$ *componentwise* $\backsim$ *to* $r$ *and a* $q'$ *such that* $p' \xrightarrow{a,\mathbf{P}^k\to\mathbf{P}} q'$ *and* $qr \backsim q'r'$.

*Processes* $p$ *and* $p'$ *are* white bisimilar, $p \leftrightarrows_{white} p'$, *if there exists a white bisimulation relation relating them.*

The definition of white bisimulation is monotone, and thus the usual basic bisimulation theory applies; *e.g.*, $\leftrightarrows_{white}$ is itself a white bisimulation. Note that $\leftrightarrows_{white}$ is a rather peculiar bisimulation. Suppose that we have the following operation, where $b$ is a fixed action, but $a$ and $\xi$ ranges over types.

$$\frac{x \xrightarrow{a,\xi} y, \quad x \xrightarrow{b,\mathbf{P}} y'}{f(x) \xrightarrow{a,\xi} y}$$

That is, $f(p)$ behaves like $p$ if $p$ has a $b$-transition at type $\mathbf{P}$ on its first move, and $\mathbf{0}$ otherwise. Consider the processes

$$\begin{array}{rcccl} p & = & (a?x.\mathbf{0}) + (a?x.x) & = & a(\lambda x.\mathbf{0}) + a(\lambda x.x) \\ p' & = & p + (a?x.f(x)) & = & p + a(\lambda x.f(x)) \end{array}$$

Clearly, $p'$ can evolve via an $a$-transition into the state $\lambda x.f(x)$, which is clearly different from both $\mathbf{0}$ and $\lambda x.x$. That is, $p$ and $p'$ are intuitively not bisimilar.

However, we have $p \leftrightarrows_{white} p'$. Clearly both of $p$'s moves match moves of $p'$, and two of $p'$s moves match moves of $p$. For any $r : \mathbf{P}$, we have $qr =_{\beta\pi} f(r)$, which is either bisimilar to $r$ or to $\mathbf{0}$ depending on whether or not $r$ can take a $b$ action at $\mathbf{P}$. In particular, the usual intuitions of bisimulation, of "staying in related states on all actions," have been lost in white bisimulation.[5]

Nonetheless, white bisimulation is a worthwhile notion. It enjoys many of the properties of bisimulation in ordinary process algebra; *e.g.*, the method of "bisimulation up to $\leftrightarrows_{white}$" works. More importantly, $\leftrightarrows_{white}$ is a good notion of process equivalence on a suitable class of languages; that is, all suitably defined operations respect $\leftrightarrows_{white}$.

**Definition 4.3** *A* White Chocolate Language *is a set of operations over a white chocolate type system, and a set of rules, subject to the following condition. For each rule* $\rho$, *let* $u$ *be either the target of* $\rho$ *(if the target has type* $\mathbf{P}$ *or* $\mathbf{P}^k$*), or the normal form of* $\mathsf{target}(\rho)$ *applied to a fresh variable* $z$ *(if the target has type* $\mathbf{P}^k \to \mathbf{P}$*). Then, we require that*

1. *If a target variable* $y$ *of type* $\mathbf{P}^k \to \mathbf{P}$ *appears in* $u$, *then it appears only once, and that occurrence is in a subterm of the form* $y(p)$ *for some term* $p$.

[5] This may also be construed as a flaw in the higher-order transition system.

*2. There are no abstractions in $u$.*

For example, if we restrict NESTLÉ to white chocolate types, we get a white chocolate language. Consider the restriction rule. Ignoring some tupling and projecting that the actual system requires, we have the rule

$$\frac{x \xrightarrow{a, \mathbf{P}^1 \to \mathbf{P}} y}{x\backslash S \xrightarrow{a, \mathbf{P}^1 \to \mathbf{P}} \lambda w.(yw)\backslash S}$$

Then $u = (\lambda w.(yw)\backslash S)z =_{\beta\pi} (yz)\backslash S$, which satisfies the definition.

**Theorem 4.4** *Let $\mathcal{L}$ be a white chocolate language. White bisimulation is a congruence for $\mathcal{L}$.*

This subsumes, *e.g.*, Thomsen's result that white bisimulation (his higher-order bisimulation) is a congruence for CHOCS; it also implies that white bisimulation is a congruence for Hennessy's somewhat more elaborate language, and for NESTLÉ/2.

# 5 Acknowledgements

We would like to asknowledge Paul Taylor for his diagram package, which was helpful for drawing the many sorts of amazing arrows used in this study.

# References

[Ace92] Luca Aceto. Eliminating junk rules from GSOS languages. (Unpublished note; to appear, probably as part of something else.), July 1992.

[BC90] Kenneth Birman and Robert Cooper. The Isis project: Real experience with a fault tolerant programming system. Technical Report 90-1138, Department of Computer Science, Cornell University, July 1990.

[BCG91] Kenneth P. Birman, Robert Cooper, and Barry Gleeson. Programming with process groups: Group and multicast semantics. Technical Report 91-1185, Department of Computer Science, Cornell University, January 1991.

[BIM88] Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced (preliminary report). In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 229–239, 1988. Also appears as MIT Technical Memo MIT/LCS/TM-345.

[Blo89] Bard Bloom. *Ready Simulation, Bisimulation, and the Semantics of CCS-Like Languages.* PhD thesis, Massachusetts Institute of Technology, August 1989.

[Hen93] Matthew Hennessy. A fully abstract denotational model for higher-order processes (extended abstract). In *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 397–408. IEEE Computer Society Press, 1993. (Full version in Sussex TR 6/92).

[Mil91] Robin Milner. The polyadic $\pi$-calculus: A tutorial. In *Proc. International Summer School on Logic an dAlgebra of Specification.* Marktoberdorf, 1991.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes I and II. *Information and Computation*, 100:1–40 and 41–77, 1992.

[Rep89] John H. Reppy. First-class synchronous operations in Standard ML. Technical Report 89-1068, Department of Computer Science, Cornell University, December 1989.

[Rep91] John H. Reppy. CML: A higher-order concurrent language. *SIGPLAN Notices*, 26(6):293–305, June 1991.

[Tho89] Bent Thomsen. A calculus of higher order communicating systems. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 142–154, 1989.

[Vaa93] Frits Vaandrager. Expressiveness results for process algebras. In de Bakker, de Roever, and Rozenberg, editors, *Semantics: Foundations and Applications*, pages 609–620. Springer-Verlag, 1993. LNCS 666. Also appears as CWI Tech Report CS-R9301.