Operational and Algebraic Semantics for Facile: A Symmetric Integration of Concurrent and Functional Programming

Sanjiva Prasad Alessandro Giacalone [†] Prateek Mishra
Department of Computer Science
The State University of New York at Stony Brook
Stony Brook, New York 11794-4400
sanjiva@sbcs.sunysb.edu, ag@sbcs.sunysb.edu, mishra@sbcs.sunysb.edu
† Author's current address is ECRC, Arabellastrasse 17, D–8000 Munich 81, F.R.G.

Abstract

We study the semantic foundations of a class of languages that combine the typed λ-calculus with process constructs for synchronised communication over typed channels. Examples include PFL, Amber, PML and Facile, a language under development at Stony Brook that supports a symmetric integration of functional and concurrent programming. Expressions in Facile are very general and may dynamically create processes and cause communication.

The main result is an extensional notion of program *behaviour* which we characterise using operational techniques. A new concept of abstraction in observation is introduced: a system is observed through a set of channels called a *window.* The concept of window is used to generalise bisimulation to indexed families of relations, thus supporting reasoning about higher-order processes whose interface may expand dynamically during execution. A definition of process and expression equivalence is introduced which takes into account both observable behaviour and resulting value. We close with a discussion on compositional reasoning using this notion of window-parametrised equivalence.

1 Introduction

Recently there has been interest in languages that combine functional and concurrent programming. One particular family combines the typed call-by-value λ-calculus with process constructs for synchronised communication over typed channels. PFL [Hol83], Amber [Car86], PML [Rep88] and Facile [GMP89b] are examples of such languages. These languages can be considered extensions of the typed λ-calculus in that: (i) they use a single binding operator – λ abstraction – and are lexically scoped; (ii) are fully higher-order, supporting both higher-order functions and processes; (iii) channels are dynamically generated and are first-class values, as in occam [INM84].

In previous work [GMP89b] we have developed Facile, a language that supports a symmetric integration of functional and concurrent programming. In [GMP89a] we have given several examples that demonstrate the expressiveness of such an integration. The main contribution of this paper is to develop the semantic foundations for integrated functional-concurrent languages. We discover an extensional notion of program *behaviour* which we characterise using operational techniques. The innovation here is to combine observable behaviour of processes with evaluation of expressions. This generalises current results in semantic frameworks for concurrency like CCS [Mil80] by providing semantics for a very general form of expression evaluation in processes (*e.g.*, expressions can create processes and cause communication of higher-order values). From the viewpoint of the typed

*This work has been partially supported by NSF CCR-8704309 and NSF CCR-8706973

λ-calculus, our treatment provides a basis for its extension with reactive input/output and indeterminacy. Our treatment here is directly in terms of a concrete language; the language includes implementation-oriented features such as channel generation which may be used to realise more abstract operations such as CCS-style restriction.

In Section 2 we introduce a "core" syntax underlying Facile and other such languages. For clarity in this presentation, the syntax is two-sorted: *expressions* and *behaviour expressions.* Following the approach taken in [Mil80], the notion of program behaviour is developed in two steps. First, in Section 3, we provide a structural operational semantics Plotkin [Plo81] in terms of a labeled reduction. The labeled transition system defines two relations: an *evaluates* relation on function expressions and a *derives* relation on behaviour expressions. Labels marking an expression reduction represent the observable effects of expression evaluation. As expression evaluation may produce observable effects, a fixed order of evaluation is imposed by the semantic definition.

Second, in Section 4, we introduce a notion of observability based on communication, following [Mil80]. Central to the definition of observability is a new concept of abstraction in observation: a system is observed through a set of channels called a *window.* A window hides "local" channels from an observer. In CCS, which has static sorts, this is accomplished by restriction. In contrast, Facile processes may use channels in the window to export local channels, both explicitly, and implicitly through higher-order objects. Thus, windows may *expand* dynamically during execution.

The question that we address in this paper is: "If we observe the executions of two programs, starting with a given interface, can we distinguish them?" Equivalence is based on a notion of *window observability*: two processes are equivalent under window W if they can *continually* simulate each other when initially observed through W. The definition of equivalence has three mutually recursive components: *behaviour equivalence*, *expression equivalence*, and equivalence between observable actions (*action equivalence*). We have chosen the (weak) bisimulation approach [Par81, Mil88] as the basis for behaviour equivalence because it is appropriate for reactive systems. Two expressions are equivalent if they reduce to equivalent values while producing *equivalent* observable behaviour. In [Mil80], observable actions are either identical or distinct. In contrast, because we have higher-order values that can be communicated, we introduce a concept of equivalent observable actions based on the notion of equivalent values. For example, two processes that output different code for the factorial function on the same output channel would be considered as performing equivalent actions under our definition.

The proof method we use extends the notion of bisimulation relations [Mil83, Par81] to families of relations indexed by windows. This makes it possible to reason about programs that expand their interface during execution, and also supports reasoning about programs with respect to different interfaces.

Section 5 reports equational rules that support reasoning about programs. Rules for reasoning about the call-by-value λ-calculus, such as those found in [Plo75], carry through unchanged. Furthermore, analogues of important properties of CCS such as the commutativity, associativity of the parallel composition and choice operators are preserved. In this sense, one may consider our framework as providing a "conservative extension" of both the call-by-value λ-calculus and CCS. In Section 6, we pose the question about using our notion of equivalence for compositional reasoning. Some of the difficulties in obtaining a general result are outlined.

Related Work

We are not aware of any formal semantics for Amber or for PML. The language PFL has been given an operational semantics but no definition of program equivalence. As noted above, our work can be seen as providing a semantic foundation for these languages.

PFL is an occam-like language expressed in a functional-style syntax using continuations, with behaviours and abstractions on them as the primary data types. Facile differs from PFL in that it integrates functional and concurrent programming symmetrically, providing a rich variety of data types, which may employ behaviours. The treatment of behaviours as a datatype is also proposed in SMoLCS [AR87], where a general algebraic framework is presented which enables the definition of concurrent calculi.

Extensions to CCS with ports as communicable values have been studied in [AZ84] and [EN86]. Astesiano and Zucca [AZ84] use indexed families of channels, whereas Engberg and Nielsen [EN86] give a higher-level treatment based on static scoping. In the latter, the bisimulation technique is not provided, making it difficult to reason about programs in which communication channels are exported outside their original restricted scope ("scope extrusion"). Recently, Milner, Parrow and Walker [MPW89] have refined the treatment in [EN86], giving it a bisimulation characterisation and simplifying the treatment of names. However, these calculi are not higher-order; they do not support abstraction or functional-style programming in general. Also, these calculi use explicit restriction, whereas Facile, Amber etc. use generative channel values, an approach which is more appropriate for programming.

There are several proposals that embed functions in a meta-level concurrent component [Mei89, Tur87]. However, no formal semantics has been given for such a combination. We also do not discuss here approaches using asynchronous communication that combine functional programming with concurrency such as those based on Kahn's model [Kah74].

Recently, Nielson [Nie89] has described a language TPL that combines CCS and the typed λ-calculus. A dynamic semantics is outlined but no definition of equivalence is provided. The language, like CCS, includes only static port names. In contrast, Facile has the notion of a channel value which may be dynamically created and exported. Bent Thomsen has described a higher-order extension to CCS called CHOCS [Tho89]. Processes are treated as values in both Facile and CHOCS. Unlike Facile, CHOCS inherits static sorts from CCS and does not include any facilities for expression evaluation. Both TPL and CHOCS exhibit dynamic binding of channel names, as in the transition:

$$(\ a!(b?y.y)\ .\ nil \mid (a?x.x)\backslash b\ )\backslash a \ \xrightarrow{\tau}\ (\ nil \mid (b?y.y)\backslash b\ )\backslash a$$

where the channel named b is "captured" in the context $[\ ]\backslash b$. Thus, a purely incidental choice of a restricted channel *name* in a program may lead to unintended results. TPL and Facile also differ in that the former is a call-by-name calculus. The combination of call-by-name and side-effect, in our case due to communication, is sometimes hard to understand. A classical example is *Jensen's device* as found in ALGOL 60.

2 Core Syntax of Facile

$\mathcal{I}$ denotes the set of all identifiers; *id* and x are representative identifiers. $\mathcal{C}$ denotes the set of all primitive constants; c is a representative constant. Υ, the set of types is defined inductively as :

$$t\ ::=\ int \mid bool \mid unit \mid t_1 \rightarrow t_2 \mid code \mid t\ chan$$

where t is a typical type. The type *unit* is a one-element type, and $t_1 \rightarrow t_2$ is the usual function type. The type t *chan* consists of channels on which values of type t may be transmitted. The type *code* is the type of a process script.

Definition 1 *:* (EXPRESSIONS) Expressions are defined inductively as :

$$\begin{array}{lll} exp & ::= & x \mid c \mid \lambda\ x\ .\ exp \mid exp_1\ exp_2 \mid \texttt{code}(\ Beh\_Exp\ ) \\ & \mid & \texttt{spawn}(\ Beh\_Exp\ ) \mid \texttt{channel}(\ t\ ) \mid exp_1\ !\ exp_2 \mid exp\ ? \end{array}$$

■

The set C includes integers, booleans *true* and *false*, a distinguished value *triv*, channel-valued constants and standard primitive functions. As in functional languages, expressions include variables,

λ-abstraction, and application. In addition, we have the following expressions: the *spawn* expression evaluates to *triv*, but has the external effect of creating a concurrently executing process with the specified behaviour. The *channel* expression evaluates to a *new* channel value. The *send* expression exp_1 ! exp_2 evaluates to *triv* and transmits the value of exp_2 on the channel given by exp_1's value. The *receive* expression *exp* ? evaluates to any value received on channel *exp*. Communication is synchronised as in CCS. The *code* constructor packages a behaviour expression into a value called a process script, thus making behaviours first-class values.

Definition 2 : (BEHAVIOUR EXPRESSIONS)

$$\begin{array}{lcl} Beh\_Exp & ::= & \texttt{terminate} \mid \texttt{activate}\ exp \\ & \mid & Beh\_Exp_1 \parallel Beh\_Exp_2 \mid Beh\_Exp_1\ \%\ Beh\_Exp_2 \end{array}$$

■

`terminate` is the behaviour expression signifying inaction. Parallel composition of behaviour expressions is a behaviour expression. The *activate* behaviour expression connects the world of behaviour expressions to that of expressions; the expression *exp* is evaluated for its effects finally yielding a process script, i.e. a *code* value. The current process is then replaced by one with behaviour specified in that process script. The "%" constructor defines the mutually exclusive execution of Beh_Exp_1 or Beh_Exp_2. This non-deterministic selection of an alternative is similar to the CCS sum operator.

An expression context is an expression with "holes" or positions in it where any appropriately typed sub-expression may be inserted. Contexts are denoted as $C[.]$, and $C[e]$ is an expression where the holes in context $C[.]$ has been filled by expression e. Similarly a behaviour context is a well-typed behaviour expression with holes in it.

3 Dynamic Semantics

Definition 3 : (VALUES) *Val*, syntactic values, is a subset of *exp* that comprises :

$$c, \quad \lambda x.exp, \quad \texttt{code}(\ Beh\_Exp\ )$$

where each expression is closed and well-typed. v is a typical value. ■

A *sort* is a set of typed channels. $\mathcal{S}$ denotes the universal set of all possible typed channels. $\mathcal{P}(\mathcal{S})$ denotes the powerset of $\mathcal{S}$. $\mathcal{S}_t$ denotes the subset of $\mathcal{S}$ consisting of channels on which values of type t can be communicated. $\mathcal{K}$ is a typical subset of $\mathcal{S}$, and k is a typical channel in $\mathcal{S}$. We say a value $v \in Val$ is transmittable on channel k if v is a value of type t and k is a channel of type *t chan*. In this paper, we will assume that whenever we talk of a set of channels, there are unboundedly many channels of each type outside that set.

Definition 4 : (LABELS) *Comm*, the set of communication labels is defined as :

$$Comm \;=\; \{\ k(v),\ \overline{k(v)}\ \mid\ \exists t.(k \in \mathcal{S}_t,\ v \in Val,\ \emptyset \vdash\ v\ :\ t)\ \}$$

$\mathcal{L}$, the set of labels is defined as :

$$\mathcal{L} \;=\; Comm\ \cup\ \{\tau\}\ \cup\ \{\Phi(B) \mid B \in Beh\_Exp\ \}$$

■

The labels $k(v)$ and $\overline{k(v)}$ represent, respectively, a potential input and output action of a value v over a channel k. The value v should be transmittable on channel k. $k(v)$ and $\overline{k(v)}$ are called *complementary labels.* The τ label, imported from CCS, represents a "hidden" or internal atomic

1. Function Application

(a) $$\frac{\mathcal{K}, e_1 \overset{\ell}{\hookrightarrow} \mathcal{K}', e_1'}{\mathcal{K}, e_1\ e_2 \overset{\ell}{\hookrightarrow} \mathcal{K}', e_1'\ e_2}$$

(b) $$\frac{\mathcal{K}, e_2 \overset{\ell}{\hookrightarrow} \mathcal{K}', e_2'}{\mathcal{K}, v\ e_2 \overset{\ell}{\hookrightarrow} \mathcal{K}', v\ e_2'}$$

(c) $$\frac{\Box}{\mathcal{K}, (\lambda x.e)\ v \overset{\tau}{\hookrightarrow} \mathcal{K}, e[x \mapsto v]}$$

2. Send

(a) $$\frac{\mathcal{K}, e_1 \overset{\ell}{\hookrightarrow} \mathcal{K}', e_1'}{\mathcal{K}, e_1 \mathbin{!} e_2 \overset{\ell}{\hookrightarrow} \mathcal{K}', e_1' \mathbin{!} e_2}$$

(b) $$\frac{\mathcal{K}, e_2 \overset{\ell}{\hookrightarrow} \mathcal{K}', e_2'}{\mathcal{K}, k \mathbin{!} e_2 \overset{\ell}{\hookrightarrow} \mathcal{K}', k \mathbin{!} e_2'}$$

(c) $$\frac{\Box}{\mathcal{K}, k \mathbin{!} v \overset{k(v)}{\hookrightarrow} \mathcal{K}, \mathit{triv}}$$

provided $k \in \mathcal{K}$.

3. Receive

(a) $$\frac{\mathcal{K}, e \overset{\ell}{\hookrightarrow} \mathcal{K}', e'}{\mathcal{K}, e\ ? \overset{\ell}{\hookrightarrow} \mathcal{K}', e'\ ?}$$

(b) $$\frac{\Box}{\mathcal{K}, k\ ? \overset{k(v)}{\hookrightarrow} \mathcal{K}, v}$$

provided $k \in \mathcal{K}$, value v transmittable on k.

4. Channel Creation

$$\frac{\Box}{\mathcal{K}, \mathtt{channel}(t) \overset{\tau}{\hookrightarrow} \mathcal{K} \cup \{k\}, k}$$

where $k \notin \mathcal{K}$ and $k \in \mathcal{S}_t$.

5. Process Creation

$$\frac{\Box}{\mathcal{K}, \mathtt{spawn}(B) \overset{\Phi(B)}{\hookrightarrow} \mathcal{K}, \mathit{triv}}$$

6. Activate

(a) $$\frac{\mathcal{K}, e \overset{\ell}{\hookrightarrow} \mathcal{K}', e'}{\mathcal{K}, \mathtt{activate}\ e \overset{\ell}{\twoheadrightarrow} \mathcal{K}', \mathtt{activate}\ e'}$$

provided $\ell \neq \Phi(B)$.

(b) $$\frac{\mathcal{K}, e \overset{\Phi(B)}{\hookrightarrow} \mathcal{K}', e'}{\mathcal{K}, \mathtt{activate}\ e \overset{\tau}{\twoheadrightarrow} \mathcal{K}', (\mathtt{activate}\ e' \parallel B)}$$

(c) $$\frac{\Box}{\mathcal{K}, \mathtt{activate\ code}(\ B\ ) \overset{\tau}{\twoheadrightarrow} \mathcal{K}, B}$$

7. Concurrent Composition

(a) $$\frac{\mathcal{K}, B_1 \overset{\ell}{\twoheadrightarrow} \mathcal{K}', B_1'}{\mathcal{K}, B_1 \parallel B_2 \overset{\ell}{\twoheadrightarrow} \mathcal{K}', B_1' \parallel B_2}$$

(b) $$\frac{\mathcal{K}, B_2 \overset{\ell}{\twoheadrightarrow} \mathcal{K}', B_2'}{\mathcal{K}, B_1 \parallel B_2 \overset{\ell}{\twoheadrightarrow} \mathcal{K}', B_1 \parallel B_2'}$$

8. Complementary Communication

$$\frac{\begin{array}{c}\mathcal{K}, B_1 \overset{\ell}{\twoheadrightarrow} \mathcal{K}', B_1' \\ \mathcal{K}, B_2 \overset{\ell'}{\twoheadrightarrow} \mathcal{K}', B_2'\end{array}}{\mathcal{K}, B_1 \parallel B_2 \overset{\tau}{\twoheadrightarrow} \mathcal{K}', B_1' \parallel B_2'}$$

where ℓ and ℓ' are complementary labels.

9. Alternative

(a) $$\frac{\mathcal{K}, B_1 \overset{\ell}{\twoheadrightarrow} \mathcal{K}', B_1'}{\mathcal{K}, B_1 \mathbin{\%} B_2 \overset{\ell}{\twoheadrightarrow} \mathcal{K}', B_1'}$$

(b) $$\frac{\mathcal{K}, B_2 \overset{\ell}{\twoheadrightarrow} \mathcal{K}', B_2'}{\mathcal{K}, B_1 \mathbin{\%} B_2 \overset{\ell}{\twoheadrightarrow} \mathcal{K}', B_2'}$$

Figure 1: Dynamic Semantics

action, such as a communication between two component processes in a system. The special $\Phi(B)$ label, generated by a *spawn* expression, is used to mark the creation of a process executing code B. This label enables us to provide a concise treatment of process creation.

$Bcon \subseteq \mathcal{P}(\mathcal{S}) \times Beh\_exp$ is the set of behaviour configurations of the form ($\mathcal{K}$, B) where $\mathcal{K}$ is a sort that records channels already in existence, and B is a behaviour expression. As mentioned, we assume we can always find channels other than those already in $\mathcal{K}$. $\Gamma \in Bcon$ denotes a typical behaviour configuration.

$Econ \subseteq (\mathcal{P}(\mathcal{S}) \times exp)$, is the set of expression configurations of the form ($\mathcal{K}$, e) where $\mathcal{K}$ is a sort and e is an expression. $\gamma \in Econ$ denotes a typical expression configuration.

$\hookrightarrow \subseteq Econ \times \mathcal{L} \times Econ$ is the "evaluates" relation. $\mathcal{K}, e \overset{\ell}{\hookrightarrow} \mathcal{K}', e'$ is to be read as "one evaluation step of expression e, given the sort set $\mathcal{K}$, results in expression e' and sort set $\mathcal{K}'$, on the occurrence of event ℓ".

$\longrightarrow\!\!\!\!\gg \subseteq Bcon \times \mathcal{L} \times Bcon$ is the "derives" relation. $\mathcal{K}, B \overset{\ell}{\longrightarrow\!\!\!\!\gg} \mathcal{K}', B'$ is to be read as "Given the sort set $\mathcal{K}$, behaviour expression B derives behaviour expression B' with sort set $\mathcal{K}'$, on the occurrence of event ℓ".

Labels ℓ and ℓ' range over the set of labels $\mathcal{L}$. The letter e, possibly subscripted, stands for any expression, v for values, x for identifiers, k for channel values, and B, P, Q, R for behaviour expressions.

The "evaluates" and "derives" relations are defined inductively in terms of inference rules. In Figure 1, we present the rules for "core" Facile, omitting here the evaluation rules induced by the constants in $\mathcal{C}$.

4 Equivalences

We extend the bisimulation treatment in three ways: (1) we extend the use of the bisimulation technique to deal with expression evaluation and relate expression evaluation to process execution; (2) We extend the bisimulation technique to deal with higher-order values, *i.e.* λ-abstractions and process code; (3) We make windows an explicit parameter to our reasoning about computational behaviour, and obtain window-indexed families of relations on higher-order processes, expressions and communication actions. Indexed families of relations have previously been used for proving context dependent equivalence properties [Lar86]. Our use of indexing is not contextual, but rather with respect to an interface for observation.

Windows

A *window* is a set of channels, typically written W. When observing a system through a window, we permit the communication of values only over channels in that window. Name scoping and hiding of channels in programs is achieved through the twin mechanisms of λ-binding of names and windowing. Windows may expand dynamically to include any non-window channels that may appear in a communicated value. We then observe the new configuration of the system through this new window. This is analogous to "scope extrusion" in ECCS [EN86] and the π-calculus [MPW89], when restricted channels are communicated beyond their original scope.

The function $local(X, W) = CV(X) - W$ yields the set of channel valued constants appearing in program X which are "local" to it with respect to window W. X may be an expression, a behaviour expression or a context. Appendix A contains the definitions for $CV(X)$, the channel-valued constants in X, and $CC(\alpha)$, the channel of communication of a label. The observer should not "know" these local channel-valued constants, unless and until these values are explicitly made visible by the system via a send action.

Observations

Our notion of observability is only in terms of communication. The transitions of a system are visible only if the observer communicates with the system. τ actions are internal and hence not observable. Nor are events corresponding to Φ labels observable because they are discharged to τ labels. We define $Obs = Comm \cup \{\epsilon\}$. ϵ stands for the null string. α ranges over Obs.

An action α is called *W-secure* with respect to configuration $\mathcal{K}, B$ if $\alpha = k(v)$ implies (i) $local(B, W) \cap CV(v) = \emptyset$ and (ii) $CV(v) \subseteq \mathcal{K}$. The observer process must behave as a component process in a system would; it may only send values in which all the channel-valued constants exist, and are not local to the process under observation. W-security is necessary to avoid the analogue of name-capture during "scope intrusion" (*cf* [MPW89]). We would prefer to deal with all α, avoiding name capture by (dynamically) renaming all local channels. However, while this approach is possible in an abstract calculus, in practice it may not be acceptable or even possible in a physically distributed system of processes. Our restriction is somewhat analogous to the restriction on the free variables of M placed by Church in his original definition of substitution $[M/x]N$.

We confine our interest to closed terms, namely those with no free variables. Our definitions of equivalence can be extended to open terms in the standard way, *i.e.* by considering every ground instance of the open terms, obtained by substituting for each free variable an appropriately typed value. When reasoning about programs we are interested only in configurations $\mathcal{K}$, B and windows W where "$\mathcal{K}$ *respects* window W and program B", *i.e.* $CV(B) \subseteq \mathcal{K}$ and $W \subseteq \mathcal{K}$. This reflects our intuition that channel constants that appear in a program must already exist, having been created previously. An interface to a system is meaningful only when its channels exist.

Fact 1 *:* If α is W'-secure with respect to $\mathcal{K}, B$ and $W' \cap local(B, W) = \emptyset$, then α is also W-secure with respect to $\mathcal{K}, B$. A special case is when $W' \subseteq W$. ■

Fact 2 *:* If $\alpha \in Obs$ is W-secure with respect to $\mathcal{K}, B$ and $k \notin CV(B)$, then α is also $W \cup \{k\}$-secure with respect to $\mathcal{K} \cup \{k\}, B$.
Conversely, if $\alpha \in Obs$ is $W \cup \{k\}$-secure with respect to $\mathcal{K} \cup \{k\}, B$ and $k \notin CV(B)$, then α is also W-secure with respect to $\mathcal{K}, B$ if $\alpha = d(v)$ implies $k \notin CV(v)$. ■

Fact 1 says that if an action is secure with respect to a window and a behaviour configuration, it is also secure when using another window, provided this second window does not expose channels hidden with respect to the first window. Fact 2 describes when we can add or delete channels from the set of existing channels and the window, without affecting the security of an action.

4.1 Observable Derivations

Definition 5 *:* (OBSERVABLE DERIVATION) The relation

$$\longrightarrow\!\triangleright \subset (\ Bcon \times \mathcal{P}(\mathcal{S})\ ) \times Obs \times (\ Bcon \times \mathcal{P}(\mathcal{S})\ )$$

called the *observable derivation over window W*, is defined as

$$\Gamma,\ W \ \overset{\alpha}{\longrightarrow\!\triangleright}\ \Gamma',\ W' \text{ iff}$$

1. $\Gamma \ \overset{\tau^*\alpha\tau^*}{\longrightarrow\!\!\!\gg}\ \Gamma'$ and
2. α is W-secure with respect to Γ and
3. $\begin{cases} \text{if} \quad \alpha = \epsilon \quad \text{then} \quad W' = W \\ \text{if} \quad \alpha \in Comm \quad \text{then} \quad CC(\alpha) \subseteq W \ \text{ and } W' = W \cup CV(\alpha) \end{cases}$

■

We have overloaded the symbol $\longrightarrow\!\!\!\gg$ in Definition 5 to denote the reflexive transitive closure of the "derives" relation denoted by $\longrightarrow\!\!\!\gg$. A configuration can make arbitrarily many τ moves, which are not observable. Further, the system cannot communicate with the external world over channels not in W. When a value containing non-window channels is communicated via a window channel, we expand the window by adding *all* those channels. This is a conservative guess about the *potential* for subsequent communications on these channels between a process and its environment. At the implementation level, this yields a good rule of thumb: that it is safe to garbage-collect channels created in the past that do not appear in the window or in the current program, because other processes cannot be using them. Some facts about $\longrightarrow\!\!\triangleright$:

Fact 3 *:* Suppose $\mathcal{K} \supseteq CV(B) \cup W$ and $(\mathcal{K}, B),\ W \overset{\alpha}{\longrightarrow\!\!\triangleright} (\mathcal{K}', B'),\ W'$. Then $CC(\alpha) \subseteq CV(B) \cap W$. *I.e.* if $h \notin CV(B)$ and $(\mathcal{K}, B),\ W \overset{\alpha}{\longrightarrow\!\!\triangleright} (\mathcal{K}', B'),\ W'$, then $h \notin CC(\alpha)$. ■

Fact 4 *:* Suppose $CV(B) \cup W \subseteq \mathcal{K}$ and $(\mathcal{K}, B),\ W \overset{\alpha}{\longrightarrow\!\!\triangleright} (\mathcal{K}', B'),\ W'$. Then, $CV(B') \cup W' \subseteq \mathcal{K}'$. ■

Proposition 5 *:* Suppose $CV(B) \cup W \subseteq \mathcal{K}$ and $(\mathcal{K}, B),\ W \overset{\alpha}{\longrightarrow\!\!\triangleright} (\mathcal{K}', B'),\ W'$.
Then for any $\mathcal{D}$ such that $\mathcal{D} \cap (\mathcal{K}' - \mathcal{K}) = \emptyset$: $(\mathcal{K} \cup \mathcal{D}, B),\ W \overset{\alpha}{\longrightarrow\!\!\triangleright} (\mathcal{K}' \cup \mathcal{D}, B'),\ W'$. ■

Fact 6 *:* Suppose (i) $CV(B) \cup W \subseteq \mathcal{K}$, (ii) $(\mathcal{K}, B),\ W \overset{\alpha}{\longrightarrow\!\!\triangleright} (\mathcal{K}', B'),\ W'$. and (iii) let h be such that $h \notin CV(B)$ and $h \in CV(B')$. Then, *either* h is generated *or* it is received in α. That is, (1) $h \notin \mathcal{K}$ iff $\alpha = d(v)$ implies $h \notin CV(\alpha)$, and (2) $h \in \mathcal{K}$ iff $\alpha = d(v)$ implies $h \in CV(\alpha)$. ■

Fact 7 *:* Suppose (i) $CV(B) \cup W \subseteq \mathcal{K}$, (ii) $(\mathcal{K} \cup \mathcal{D}, B),\ W \overset{\alpha}{\longrightarrow\!\!\triangleright} (\mathcal{K}' \cup \mathcal{D}, B'),\ W'$. and (iii) $\mathcal{D} \cap CV(B) = \emptyset$. Then, $\alpha = k(v)$ implies $\mathcal{D} \cap CV(v) = \emptyset$. ■

Proposition 8 *:* Suppose (i) $CV(B) \cup W \subseteq \mathcal{K}$, (ii) $(\mathcal{K}, B),\ W \overset{\alpha}{\longrightarrow\!\!\triangleright} (\mathcal{K}', B'),\ W'$. and (iii) $\mathcal{D}$ is a set such that $\mathcal{D} \cap CV(B) = \emptyset$ and $\mathcal{D} \cap CV(\alpha)$.
Then, $(\mathcal{K} - \mathcal{D}, B),\ W - \mathcal{D} \overset{\alpha}{\longrightarrow\!\!\triangleright} (\mathcal{K}' - \mathcal{D}, B'),\ W' - \mathcal{D}$. ■

Definition 6 *:* (Isomorphic Channel Renaming) An isomorphism $\varsigma : \mathcal{S} \rightarrow \mathcal{S}$ is called an isomorphic channel renaming if it respects the types of channels. That is $\varsigma(\mathcal{S}_t) = \mathcal{S}_t$. ■

We also define renamings that are not isomorphic using the notation $\{h/k\}$ for "k is replaced by h", where k and h are channel-valued constants of the same type.

Proposition 9 *:* If $(\mathcal{K}, B),\ W \overset{\alpha}{\longrightarrow\!\!\triangleright} (\mathcal{K}', B'),\ W'$ and ς is an isomorphic channel renaming. Then, $(\mathcal{K}\varsigma, B\varsigma),\ W\varsigma \overset{\alpha\varsigma}{\longrightarrow\!\!\triangleright} (\mathcal{K}'\varsigma, B'\varsigma),\ W'\varsigma$. ■

4.2 Window-Parametrised Relations

We now formulate notions of equivalence for behaviour configurations, expressions, and actions. These notions will be parametrised by the window of observation W. Accordingly, we define three kinds of window-to-relation maps:
$B\_Bis = \mathcal{P}(S) \rightarrow (Beh\_Exp \times Beh\_Exp)$. $\mathcal{B} \in B\_Bis$ is a typical function from windows to relations on behaviour expressions. $F\_Bis = \mathcal{P}(S) \rightarrow (exp \times exp)$. $\mathcal{F} \in F\_Bis$ is a typical function from windows to relations on expressions. $A\_Bis = \mathcal{P}(S) \rightarrow (Obs \times Obs)$. $\mathcal{A} \in A\_Bis$ is a typical function from windows to relations on actions. Let $Bisim = (\ B\_Bis \times F\_Bis \times A\_Bis\ )$.

We will write $\mathcal{B}(W)$ (respectively $\mathcal{F}(W)$, $\mathcal{A}(W)$) to denote the relation obtained by fixing W in $\mathcal{B}$ (respectively $\mathcal{F}$, $\mathcal{A}$).

There is a natural ordering on these triples of window-to-relation maps, namely component-wise, point-wise, subset inclusion. This ordering $\preceq$ on *Bisim* is in fact a complete lattice. From standard fixpoint theory, any monotonic function ψ has a maximal fixed point $\nu\psi$:

$$\nu\psi \;=\; \bigsqcup\{x \mid \psi(x) = x\} \;=\; \bigsqcup\{x \mid x \;\preceq\; \psi(x)\}$$

We define three mutually recursively properties $\star_{\mathcal{B}}$, $\star_{\mathcal{A}}$, and $\star_{\mathcal{F}}$ below. These properties simultaneously induce a transformation Ψ which is monotonic on *Bisim*. We will show that the W-components of its maximal fixed point — written as $\overset{W}{\approx}$ on behaviour expressions, $\overset{W}{\sim}$ on expressions, and $\overset{W}{\asymp}$ on actions — are equivalence relations.

Given a mapping $\mathcal{B} \in B\_Bis$, and an isomorphic channel renaming ς, we can form a mapping $\mathcal{B}_\varsigma$ such that $\langle P,Q\rangle \in \mathcal{B}(W)$ iff $\langle P\varsigma, Q\varsigma\rangle \in \mathcal{B}_\varsigma(W\varsigma)$. We say that a mapping $\mathcal{B} \in B\_Bis$ is *closed under isomorphic channel renaming* if for every isomorphic channel renaming ς: $\langle P,Q\rangle \in \mathcal{B}(W)\, iff\, \langle P\varsigma, Q\varsigma\rangle \in \mathcal{B}(W\varsigma)$. Similarly for mappings in *F_Bis* and *A_Bis*. It is natural to expect our equivalences to be closed under isomorphic channel renaming. While programs in our language can explicitly mention channel constants, our notion of equivalence should not depend on the particular identity of any channel constant.

Another spurious distinction between programs arises from examining whether one program can generate a *particular* channel that the other cannot, because the latter was already using this as a local channel. If we allow renaming of local channels at run-time, as in [MPW89], there would be no problem. As discussed before, this may not be viable in real systems. Instead, we insist that in comparing the behaviour of two programs P and Q through window W, that P will not generate any channels that are local to Q and vice versa. We do so by starting with configurations where the set of existing channels $\mathcal{K} \supseteq CV(P) \cup CV(Q) \cup W$.

Definition 7 : Suppose we are given a pair $\langle \mathcal{B}, \mathcal{A}\rangle \in B\_Bis \times A\_Bis$. Also given are B_1, B_2, W and $\mathcal{K}$ such that $CV(B_1) \cup CV(B_2) \cup W \subseteq \mathcal{K}$. We say that from the initialisation of window W and sort $\mathcal{K}$ the derivatives of B_1 and B_2 under every $\mathcal{A}$-related simulation are $\mathcal{B}$-related, if the following holds:

$\forall \alpha \in Obs$ such that α is W-secure with respect to $\mathcal{K}, B_1$ and $\mathcal{K}, B_2$:

1. $\mathcal{K}, B_1,\ W \xrightarrow{\alpha}\!\!\triangleright\ \mathcal{K}_1, B_1',\ W'$ implies $\exists \mathcal{K}_2, B_2'$ and α' such that
   - i. $(\mathcal{K}, B_2),\ W \xrightarrow{\alpha'}\!\!\triangleright\ (\mathcal{K}_2, B_2'),\ W'$
   - ii. $\langle \alpha,\ \alpha'\rangle \in \mathcal{A}(W_\alpha)$, where $CV(\alpha') = CV(\alpha) = W_\alpha$
   - iii. $\langle B_1',\ B_2'\rangle \in \mathcal{B}(W')$
2. $\mathcal{K}, B_2,\ W \xrightarrow{\alpha}\!\!\triangleright\ \mathcal{K}_2, B_2',\ W'$ implies $\exists \mathcal{K}_1, B_1'$ and α' such that
   - i. $(\mathcal{K}, B_1),\ W \xrightarrow{\alpha'}\!\!\triangleright\ (\mathcal{K}_1, B_1'),\ W'$
   - ii. $\langle \alpha',\ \alpha\rangle \in \mathcal{A}(W_\alpha)$, where $CV(\alpha') = CV(\alpha) = W_\alpha$
   - iii. $\langle B_1',\ B_2'\rangle \in \mathcal{B}(W')$

We depict this property as
$$\boxed{\begin{array}{lccc} & \mathcal{K}, B_1 & \longrightarrow & . \\ W \vdash & & \updownarrow \mathcal{A} & \updownarrow \mathcal{B} \\ & \mathcal{K}, B_2 & \longrightarrow & . \end{array}}$$
■

This property says that if we start with an initial sort $\mathcal{K}$ and window W then every observable derivation of B_1 can be simulated by B_2 performing an $\mathcal{A}$-related action, with the derivatives being related by $\mathcal{B}$, and symmetrically for B_2. For our notion of behaviour equivalence, we will be therefore interested in mappings $\mathcal{B}$ and $\mathcal{A}$ such that whenever $\langle B_1, B_2\rangle \in \mathcal{B}(W)$, the property

$$\boxed{W \vdash \begin{tikzcd} \mathcal{K},B_1 \arrow[r] \arrow[r, phantom, shift right=4ex, "\updownarrow \mathcal{A}"] & . \arrow[d, leftrightarrow, "\mathcal{B}"] \\ \mathcal{K},B_2 \arrow[r] & . \end{tikzcd}}$$ holds for appropriate initialisations.

4.3 Bisimiles

We are interested in defining our notion of equivalence in terms of a triple $\langle \mathcal{B}, \mathcal{F}, \mathcal{A}\rangle$ that simultaneously satisfy the properties $\star_{\mathcal{B}}$, $\star_{\mathcal{F}}$ and $\star_{\mathcal{A}}$ given below.

Definition 8 : (PROPERTY $\star_{\mathcal{B}}$)
$\langle B_1,\ B_2\rangle \in \mathcal{B}(W)$ iff

$\forall \mathcal{K}$ that respect B_i, W $(i = 1, 2)$: $$\boxed{W \vdash \begin{tikzcd} \mathcal{K},B_1 \arrow[r] \arrow[r, phantom, shift right=4ex, "\updownarrow \mathcal{A}"] & . \arrow[d, leftrightarrow, "\mathcal{B}"] \\ \mathcal{K},B_2 \arrow[r] & . \end{tikzcd}}$$ holds. ∎

The idea is that two behaviour expressions are indistinguishable if they can mimic each other, and in doing so never reach configurations which can be distinguished via communication. Given a window W, whenever behaviour B_1 can observably derive by action α the behaviour B_1' and new window W', then behaviour B_2 can observably derive by an equivalent action behaviour B_2', with the *same* new window W'. Further the new behaviours should be indistinguishable with respect to window W'. Note that condition (ii) requires that the two experiments α and α' be considered "equivalent", and not necessarily identical. This is because Facile admits higher-order values.

Definition 9 : (PROPERTY $\star_{\mathcal{A}}$)
For $\alpha_1,\ \alpha_2 \in \mathit{Obs}$, $\langle \alpha_1,\ \alpha_2\rangle \in \mathcal{A}(W)$ iff either (1) (Empty Actions) $\alpha_1 = \epsilon = \alpha_2$, or (2) (Communication Actions) for *some* k, *either* (2.1) $\alpha_1 = k(v_1)$ and $\alpha_2 = k(v_2)$ and $\langle v_1, v_2\rangle \in \mathcal{F}(W)$ *or* (2.2) $\alpha_1 = \overline{k(v_1)}$ and $\alpha_2 = \overline{k(v_2)}$ and $\langle v_1, v_2\rangle \in \mathcal{F}(W)$ ∎

Definition 10 : (PROPERTY $\star_{\mathcal{F}}$)
$\langle e_1^t,\ e_2^t\rangle \in \mathcal{F}(W)$, where annotation t indicates the type of the expression, iff one of the following cases holds:

1. (Values) *i.e.* $e_1 \equiv v_1^t$ and $e_2 \equiv v_2^t$
   (a) (Atomic values *i.e.* for $t \in \{\mathit{int},\ \mathit{bool},\ \mathit{unit},\ T\ \mathit{chan}\}$) $v_1 \equiv v_2$
   (b) (Higher-order values *i.e.* $t \in \{t_1 \rightarrow t_2,\ \mathit{code}\}$)
   $\forall$ *closed* $B[.^t]$: $\langle B[v_1], B[v_2]\rangle \in \mathcal{B}(W \cup CV(B[.]))$
2. (General Expressions *i.e* not both values.) for "new" channel $r \in \mathcal{S}_t$, $r \notin CV(e_1) \cup CV(e_2)$
   $\langle (r!e_1\ ;\ \texttt{terminate}),\ (r!e_2\ ;\ \texttt{terminate})\rangle \in \mathcal{B}(W \cup \{r\})$

∎

For two expressions to be equivalent they must have the same type. Expression equivalence is also parametrised by a window W, since it is dependent on behaviour equivalence. Intuitively, two expressions are considered equivalent if they yield equivalent values *and* in doing so produce equivalent external effects. Atomic values are equivalent iff they are identical. Higher-order values are equivalent if processes using them in any context are indistinguishable. In general, two expressions are equivalent if the processes evaluating them have equivalent behaviour, and that they yield the same result, if at all. Here we have used the sugared syntax $e\ \underline{;\ B}$ for **activate** $(\lambda z.\mathtt{code}(B))\ e$, where $z \notin FV(B)$. If ever $(r!e\ ;\ \texttt{terminate})$ produces the action $\overline{r(v)}$, we say that e *can evaluate* to value v.

The property $\star_{\mathcal{B}}$ defines a transformation $\Psi_{\mathcal{B}} \in Bisim \to B\_Bis$ where $\langle B_1, B_2 \rangle \in (\Psi_{\mathcal{B}}(\mathcal{B}, \mathcal{F}, \mathcal{A})) (W)$ iff

$\forall \mathcal{K}$ that respect B_i, W $(i = 1, 2)$: $\boxed{\begin{array}{lccc} & \mathcal{K}, B_1 & \longrightarrow & . \\ W \vdash & & \updownarrow \mathcal{A} & \updownarrow \mathcal{B} \\ & \mathcal{K}, B_2 & \longrightarrow & . \end{array}}$ holds. The property $\star_{\mathcal{A}}$ defines a transformation $\Psi_{\mathcal{A}} \in Bisim \to A\_Bis$ where $\langle \alpha_1, \alpha_2 \rangle \in (\Psi_{\mathcal{A}}(\mathcal{B}, \mathcal{F}, \mathcal{A})) (W)$ iff either of case 1 or 2 of $\star_{\mathcal{A}}$ holds. The property $\star_{\mathcal{F}}$ defines a transformation $\Psi_{\mathcal{F}} \in Bisim \to F\_Bis$ where $\langle e_1, e_2 \rangle \in (\Psi_{\mathcal{F}}(\mathcal{B}, \mathcal{F}, \mathcal{A})) (W)$ iff one of the cases in 1 or 2 of $\star_{\mathcal{F}}$ holds.
The transformation $\Psi \in Bisim \to Bisim$ is defined as :

$$\Psi(\mathcal{B}, \mathcal{F}, \mathcal{A}) = \langle \Psi_{\mathcal{B}}(\mathcal{B}, \mathcal{F}, \mathcal{A}), \ \Psi_{\mathcal{F}}(\mathcal{B}, \mathcal{F}, \mathcal{A}), \ \Psi_{\mathcal{A}}(\mathcal{B}, \mathcal{F}, \mathcal{A}) \rangle$$

Proposition 10 *:* Ψ is monotonic on ($Bisim$, $\preceq$). ∎

Definition 11 *:* (Maximal Fixpoint) We define $\langle \approx, \sim, \asymp \rangle = \nu\Psi$ ∎

We use the notation $\overset{W}{\approx}$, $\overset{W}{\sim}$ and $\overset{W}{\asymp}$ to denote the relations obtained as the W-components of $\approx$, $\sim$, and $\asymp$, respectively. A triple $\langle \mathcal{B}, \mathcal{F}, \mathcal{A} \rangle$ is called a *bisimile* if $\langle \mathcal{B}, \mathcal{F}, \mathcal{A} \rangle \preceq \Psi(\langle \mathcal{B}, \mathcal{F}, \mathcal{A} \rangle)$. The notion of bisimiles yields a powerful proof technique: in order to prove two constructs equivalent, we can exhibit an appropriate bisimile.

Bisimiles exhibit several interesting properties. They are closed under isomorphic channel renaming, under join, under symmetric inversion. The mappings to the identity relations is also a bisimile.

Proposition 11 *:* If $\langle \mathcal{B}, \mathcal{F}, \mathcal{A} \rangle$ is a bisimile, and ς is an isomorphic channel renaming, then so is $\langle \mathcal{B}_\varsigma, \mathcal{F}_\varsigma, \mathcal{A}_\varsigma \rangle$. ∎

Let $Id_{Beh} = \{ \langle W, \langle B, B \rangle \rangle \mid B \in Beh\_Exp \}$, $Id_{exp} = \{ \langle W, \langle e, e \rangle \rangle \mid e \in exp \}$, and $Id_{Obs} = \{ \langle W, \langle \alpha, \alpha \rangle \rangle \mid \alpha \in Obs \}$

Proposition 12 *:* $\langle Id_{Beh}, Id_{exp}, Id_{Obs} \rangle$ is a bisimile. ∎

If R is a function in B_Bis, F_Bis, or A_Bis, define its point-wise symmetric inversion $R^\dagger$ as: for all W . $R^\dagger(W) = R(W)^{-1}$ where $R(W)^{-1} = \{ \langle y, x \rangle \mid \langle x, y \rangle \in R(W) \}$.

Proposition 13 *:* If $\langle \mathcal{B}, \mathcal{F}, \mathcal{A} \rangle$ is a bisimile, then so is $\langle \mathcal{B}^\dagger, \mathcal{F}^\dagger, \mathcal{A}^\dagger \rangle$. ∎

Proposition 14 *:* For some index set J, if $\{ \langle \mathcal{B}_j, \mathcal{F}_j, \mathcal{A}_j \rangle \mid j \in J \}$ are bisimiles, then so is $\langle (\bigsqcup_{j \in J} \mathcal{B}_j), (\bigsqcup_{j \in J} \mathcal{F}_j), (\bigsqcup_{j \in J} \mathcal{A}_j) \rangle$. ∎

Given a triple $\langle \mathcal{B}, \mathcal{F}, \mathcal{A} \rangle$ we can form its *closure under sub-windows* $\langle \mathcal{B}^\downarrow, \mathcal{F}^\downarrow, \mathcal{A}^\downarrow \rangle$ as follows:
$\langle B_1, B_2 \rangle \in \mathcal{B}^\downarrow(W^\downarrow)$ iff $\exists W \supseteq W^\downarrow$ such that $\langle B_1, B_2 \rangle \in \mathcal{B}(W)$.
$\langle e_1, e_2 \rangle \in \mathcal{F}^\downarrow(W^\downarrow)$ iff $\exists W \supseteq W^\downarrow$ such that $\langle e_1, e_2 \rangle \in \mathcal{F}(W)$.
$\langle \alpha_1, \alpha_2 \rangle \in \mathcal{A}^\downarrow(W^\downarrow)$ iff $\exists W \supseteq W^\downarrow$ such that $\langle \alpha_1, \alpha_2 \rangle \in \mathcal{A}(W)$.

Proposition 15 *:* (Sub-windows) If $\langle \mathcal{B}, \mathcal{F}, \mathcal{A} \rangle$ is a bisimile and is closed under isomorphic channel renaming, then so is $\langle \mathcal{B}^\downarrow, \mathcal{F}^\downarrow, \mathcal{A}^\downarrow \rangle$. ∎

Given a triple $\langle \mathcal{B}, \mathcal{F}, \mathcal{A} \rangle$ we can form its *closure under "safe" window expansion* $\langle \mathcal{B}^\uparrow, \mathcal{F}^\uparrow, \mathcal{A}^\uparrow \rangle$ as follows:
$\langle B_1, B_2 \rangle \in \mathcal{B}^\uparrow(W^\uparrow)$ iff $\exists W, D$ such that $W \cup D = W^\uparrow$ and $local(B_i, W) \cap D = \emptyset$ and $\langle B_1, B_2 \rangle \in \mathcal{B}(W)$.
$\langle e_1, e_2 \rangle \in \mathcal{F}^\uparrow(W^\uparrow)$ iff $\exists W, D$ such that $W \cup D = W^\uparrow$ and $local(e_i, W) \cap D = \emptyset$ and $\langle e_1, e_2 \rangle \in \mathcal{F}(W)$.
$\langle \alpha_1, \alpha_2 \rangle \in \mathcal{A}^\uparrow(W^\uparrow)$ $\exists W, D$ such that $W \cup D = W^\uparrow$ and $local(\alpha_i, W) \cap D = \emptyset$ and $\langle \alpha_1, \alpha_2 \rangle \in \mathcal{A}(W)$.

Proposition 16 : (WINDOW EXPANSION) If $\langle \mathcal{B},\ \mathcal{F},\ \mathcal{A}\rangle$ is a bisimile, then so is $\langle \mathcal{B}^{\uparrow},\ \mathcal{F}^{\uparrow},\ \mathcal{A}^{\uparrow}\rangle$. ∎

Proposition 17 : If $\langle \mathcal{B},\ \mathcal{F},\ \mathcal{A}\rangle$ is a bisimile such that $Id_{Beh} \sqsubseteq_{B\_Bis} \mathcal{B}_i$, and if values $\langle v_1, v_2\rangle \in \mathcal{F}(W)$, then (i) $CV(v_1) = CV(v_2)$. (ii) For any channel r of the appropriate type: $\langle (r!v_1;\mathtt{terminate}),\ (r!v_2;\mathtt{terminate})\rangle \in \mathcal{B}(W \cup \{r\})$ ∎

Proposition 18 : If $\langle \mathcal{B},\ \mathcal{F},\ \mathcal{A}\rangle$ is a bisimile closed under isomorphic channel renaming and under sub-windows. If v_1 is a value and e_2 a non-value such that $\langle v_1, e_2\rangle \in \mathcal{F}(W)$, then (i) $\exists v_2$ such that e evaluates to v_2 and $\langle v_1, v_2\rangle \in \mathcal{F}(W)$. (ii) $\forall v_2'$ such that e can evaluate to v_2': $\langle v_1, v_2'\rangle \in \mathcal{F}(W)$. Similarly for the symmetric case. ∎

Let the composition of window-to-relation-maps be denoted using an infix $\diamond$. For example, $\mathcal{B}_1 \diamond \mathcal{B}_2$ is defined as: $\langle P, R\rangle \in \mathcal{B}_1 \diamond \mathcal{B}_2(W)$ iff $\exists Q.\langle P, Q\rangle \in \mathcal{B}_1(W)\ \ \&\ \ \langle Q, R\rangle \in \mathcal{B}_2(W)$. The definitions for $\mathcal{F}_1 \diamond \mathcal{F}_2$ and $\mathcal{A}_1 \diamond \mathcal{A}_2$ are similar.

Proposition 19 : If $\langle \mathcal{B}_i,\ \mathcal{F}_i,\ \mathcal{A}_i\rangle$ $(i = 1, 2)$ are bisimiles closed under isomorphic channel renaming and sub-windows such that $Id_{Beh} \sqsubseteq_{B\_Bis} \mathcal{B}_i$, then so is $\langle \mathcal{B}_1 \diamond \mathcal{B}_2,\ \mathcal{F}_1 \diamond \mathcal{F}_2,\ \mathcal{A}_1 \diamond \mathcal{A}_2\rangle$. ∎

Proof Outline: There are some complications in the proof. One is the asymmetry in property $\star_{\mathcal{F}}$ between the clauses used to define equivalence of expressions when both are values and otherwise. We address this difficulty using the Propositions 17 and 18.

The second complication arises for behaviour expressions. Suppose $\langle P, R\rangle \in \mathcal{B}_1 \diamond \mathcal{B}_2(W)$. Therefore $\exists Q$ such that $\langle P, Q\rangle \in \mathcal{B}_1(W)$, and $\langle Q, R\rangle \in \mathcal{B}_2(W)$. Pick any $\mathcal{K} \supseteq CV(P) \cup CV(R) \cup W$. Suppose α is W-secure with respect to $(\mathcal{K}, P)$ and $(\mathcal{K}, R)$, and $(\mathcal{K}, P), W \xrightarrow{\alpha}\!\triangleright (\mathcal{K}', P'), W'$. Now for our given Q, let $\Delta_1 = CV(Q) - (CV(P) \cup CV(R) \cup W)$. Note that our choice of $\mathcal{K}$ may not contain all the channels appearing in Q. Also α may not be W-secure with respect to $(\mathcal{K}, Q)$ because it contains some local channels of Q. Now let us consider a set Δ_2 that is isomorphic to Δ_1 such that $\Delta_2 \cap \mathcal{K}' = \emptyset$ and $\Delta_1 \cap \Delta_2 = \emptyset$. Consider the isomorphism ς that swaps channels from Δ_1 and Δ_2 and maps other channels to themselves. Since Δ_1, Δ_2 do not mention any channels from P, R, W, we get (a) $\langle P, Q\varsigma\rangle \in \mathcal{B}_1(W)$, and (b) $\langle Q\varsigma, R\rangle \in \mathcal{B}_2(W)$, since the bisimiles are closed under isomorphic channel renaming. Using Proposition 5, we can add Δ_2 across the α derivation. Now $\mathcal{K} \cup \Delta_2$ respects $P, Q\varsigma, R, W$. From (a) and the assumption, we can get an α' and Q' such that $(\mathcal{K} \cup \Delta_2, Q\varsigma), W \xrightarrow{\alpha'}\!\triangleright (\mathcal{K}_2', Q'), W'$, $\langle \alpha, \alpha'\rangle \in \mathcal{A}_1(W_\alpha)$ with $CV(\alpha) = CV(\alpha')$, and $\langle P', Q'\rangle \in \mathcal{B}_1(W')$. From (b) and the assumption, we can get an α'' and R' such that $(\mathcal{K} \cup \Delta_2, R), W \xrightarrow{\alpha''}\!\triangleright (\mathcal{K}_3', R'), W'$, $\langle \alpha', \alpha''\rangle \in \mathcal{A}_2(W_\alpha)$ with $CV(\alpha') = CV(\alpha'')$, and $\langle Q', R'\rangle \in \mathcal{B}_2(W')$. Now using Proposition 8, since Δ_2 does not appear in R or $CV(\alpha'')$, we can delete it across the α'' move. Thus we have found our desired α'', R'. The symmetric case is similar. Since $\mathcal{K}$ was any appropriate sort and α any W-secure label with respect to $(\mathcal{K}, P)$ and $(\mathcal{K}, R)$, we are through. ∎

From Propositions 12, 13, 11, 14, 15 and 19, we have shown:

Theorem 20 : (EQUIVALENCES) $\forall\ W$: the relations $\overset{W}{\approx}$, $\overset{W}{\sim}$, and $\overset{W}{\asymp}$ are equivalences. ∎

Definition 12 : (EQUIVALENCE OVER ALL WINDOWS) We write $B_1\ \approx\ B_2$, iff $\forall W$: $B_1\ \overset{W}{\approx}\ B_2$. We term such behaviour expressions as totally equivalent. Similarly if $\forall W : e_1\ \overset{W}{\sim}\ e_2$, we write $e_1\ \sim\ e_2$ and say that e_1 and e_2 are totally equivalent. ∎

5 Equivalence Properties

Using bisimiles, we obtain several equational properties, some of which are tabulated below. These include the identity, commutativity and associativity properties for parallel composition and nondeterministic choice. α-equivalence and a call-by-value form of β-equivalence also hold.

1	$B \parallel \texttt{terminate} \approx B$	IDENTITY FOR $\parallel$
2	$B_1 \parallel B_2 \approx B_2 \parallel B_1$	COMMUTATIVITY OF $\parallel$
3	$(B_1 \parallel B_2) \parallel B_3 \approx B_1 \parallel (B_2 \parallel B_3)$	ASSOCIATIVITY OF $\parallel$
4	$B \,\%\, \texttt{terminate} \approx B$	IDENTITY FOR %
5	$B_1 \,\%\, B_2 \approx B_2 \,\%\, B_1$	COMMUTATIVITY OF %
6	$(B_1 \,\%\, B_2) \,\%\, B_3 \approx B_1 \,\%\, (B_2 \,\%\, B_3)$	ASSOCIATIVITY OF %
7	$B \,\%\, B \approx B$	IDEMPOTENCE OF %
8	$\texttt{activate code}(B) \approx B$	ACTIVATION IS SILENT
9	$\lambda x.e \sim \lambda z.(e[x \mapsto z])$ provided $z \notin FV(e)$	α-CONVERTIBILITY
10	$(\lambda x.e)\ v \sim e[x \mapsto v]$	β-SUBSTITUTION OF VALUES

We also obtain the following properties about equivalences:

Proposition 21 : (SUB-WINDOWS) $B_1 \overset{W}{\approx} B_2$ implies $\forall W' \subseteq W: B_1 \overset{W'}{\approx} B_2$
$e_1 \overset{W}{\sim} e_2$ implies $\forall W' \subseteq W: e_1 \overset{W'}{\sim} e_2$ ∎

Proposition 22 : (WINDOW ENLARGEMENT) For any sort $\mathcal{D}$:
$B_1 \overset{W}{\approx} B_2$ implies $B_1 \overset{W \cup \mathcal{D}}{\approx} B_2$ if $local(B_i, W) \cap \mathcal{D} = \emptyset$.
$e_1 \overset{W}{\sim} e_2$ implies $e_1 \overset{W \cup \mathcal{D}}{\sim} e_2$ if $local(e_i, W) \cap \mathcal{D} = \emptyset$. ∎

Proposition 21, obtained from Proposition 15, states that if two behaviours are equivalent with respect to any window W then they are equivalent with respect to any window that is a subset of W. Likewise, W-equivalent expressions are also equivalent with respect to any subset of W. This proposition shows the interrelationship between members of the window-parametrised equivalence families — the functions $\approx$ and $\sim$ are anti-monotonic on windows. The analogous proposition in CCS is $P \approx Q$ implies $P\backslash L \approx Q\backslash L$. Conversely, in Proposition 22, obtained from Proposition 16, we see it is permissible for inaccessible channel values to be added to windows, without altering equivalence.

Proposition 23 : (CHANNEL RENAMING) Suppose $B_1 \overset{W}{\approx} B_2$, and let h, k be any two channels of the same type such that $h, k \notin local(B_i, W)$. Then, $B_1\{h/k\} \overset{W\{h/k\}}{\approx} B_2\{h/k\}$. ∎

6 Towards Compositionality

Although the bisimile-based approach seems the correct approach for describing the semantics of Facile programs, it is often difficult to discover appropriate bisimiles. We wish to reason in a compositional style using intuitive equational rules as "tools" for program transformation, optimisation and for demonstrating program correctness.

We therefore wish to investigate when in a given context we can replace an expression or behaviour expression by an equivalent one while preserving the original program's behaviour. It is apparent that the conditions on any such "modularity" or "congruence-like" rules must be parametrised by windows. Thus, we seek to answer questions of the form: "if $F1$ and $F2$ are W-equivalent, then for context $C[.]$ and window W', under what conditions are $C[F1]$ and $C[F2]$ W'-equivalent ?" Of

course, we are interested in finding the weakest restrictions on $C[.]$, W, W' sufficient to ensure the soundness of the rules.
However, this is difficult in general. For some contexts, we can obtain conditions without much difficulty, *e.g.* Propositions 24 and 25.

Proposition 24 *:* (ALTERNATIVES)

$$B_1 \overset{W}{\approx} B_2 \text{ implies } B_1\%C \overset{W}{\approx} B_2\%C$$

if B_1, B_2 are both stable or both not stable configurations. ∎

A configuration B is called stable if $\nexists B'$ such that $B \overset{\tau}{\longrightarrow\!\!\!\!\!\rightarrow} B'$. Likewise we call an expression e stable if $\nexists \mathcal{K}', e'$ such that $\mathcal{K}, e \overset{\tau}{\hookrightarrow} \mathcal{K}', e'$. The condition in Proposition 24 is conjectured sufficient (*cf* CCS observational congruence [Mil88]) to ensure that one process does not choose an alternative silently which the other cannot.

Proposition 25 *:* (PARALLEL COMPOSITION)

$$B_1 \overset{W}{\approx} B_2 \text{ and } C_1 \overset{W}{\approx} C_2 \text{ implies } B_1||C_1 \overset{W'}{\approx} B_2||C_2$$

for windows W, W' if: (i) $local(C_1, W) \cap local(B_1, W) = \emptyset$ and $local(C_2, W) \cap local(B_2, W) = \emptyset$, and (ii) $W' \cap (\ local(B_i, W) \cup local(C_i, W)\ ) = \emptyset$. ∎

It is more difficult to obtain such conditions for expression contexts, *i.e.*

"If $e_1 \overset{W}{\sim} e_2$, then when is $C[e_1] \overset{W}{\sim} C[e_2]$ for a typed context $C[.]$?"

One obvious problem is when the e_i are placed in the head position of a % context. Thus we need a condition *(A)* on both being stable or both being not stable. Another problem arises because of the possibility of multiple copies of e_i evaluating simultaneously. Unobserved interaction on local channels between these different copies may be eliminated by assuming *(B)* that $CV(e_i) \subseteq W$. We conjecture that conditions *(A)* and *(B)* are sufficient. The main problem in a proof arises when $C[e_i]$ is a value – *i.e.* a code object or a λ-abstraction.

The two results that are difficult to show are:

(1) $$CV(B_1) = CV(B_2) \subseteq W \text{ and } B_1 \overset{W}{\approx} B_2 \text{ implies } \mathbf{code}(B_1) \overset{W}{\sim} \mathbf{code}(B_2).$$

(2) $$CV(e_1) = CV(e_2) \subseteq W \text{ and } \forall v^{t_1} : \quad e_1[x \mapsto v] \overset{W \cup CV(v)}{\sim} e_2[x \mapsto v] \text{ implies } \lambda x.e_1 \overset{W}{\sim} \lambda x.e_2.$$

The essential problem here is that for values we need to show that they yield equivalent behaviour when placed in *any* behaviour context. We conjecture that it is sufficient to show that code values cannot be distinguished unless their activations yield different behaviours, and that λ-abstractions cannot be distinguished unless their applications are distinguishable. This characterisation of equivalent values would be more intuitive from the functional programming viewpoint. We must acknowledge one of the referees for alerting us to a point that exposed an error in a proposed "proof".

The difficulty arises from the the rich variety of behaviour contexts. For example, a context may activate a code object v_1 and send this process a value containing v_1 nested within it. Any approach based on the structure of contexts, such as bisimile and arguments about homologous positions [Plo75], is doomed to failure. This is because the derivatives of equivalent expressions may not bear any similarity in their structure. Therefore, we need to follow the reduction sequences of $B[v_1]$ and $B[v_2]$, where $B[.]$ is any behaviour context, marking the positions where v_1 and v_2 or their *residuals* appear, and arguing that certain relationships about the capability to communicate on the same channels are maintained between their respective derivatives in any simulation. However, structure-based approaches may be successful for restricted versions of the language – *e.g.* where expressions cannot spawn processes.

References

[AR87] Egidio Astesiano and Gianna Reggio. *SMoLCS-Driven Concurrent Calculi.* In *LNCS 249 : TAPSOFT '87*, pages 169–201, Springer-Verlag, Berlin, 1987.

[AZ84] Egidio Astesiano and Elena Zucca. *Parametric Channels via Label Expressions in CCS.* Technical Report, Universita di Genova, January 1984.

[Car86] Luca Cardelli. *Amber.* In Cousineau, Curien, and Robinet, editors, *LNCS 242: Combinators and Functional Programming Languages*, pages 21–47, Springer-Verlag, 1986.

[EN86] U. Engberg and M. Nielsen. *A Calculus of Communicating Systems with Label Passing.* Technical Report DAIMI PB-208, Aarhus University Computer Science Department, 1986.

[GMP89a] A. Giacalone, P. Mishra, and S. Prasad. *FACILE: A Symmetric Integration of Concurrent and Functional Programming. International Journal of Parallel Processing*, 18(2):121–160, April 1989.

[GMP89b] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. *FACILE : A Symmetric Integration of Concurrent and Functional Programming.* In J. Diaz and F. Orejas, editors, *LNCS 352 : TAPSOFT '89*, pages 184–209, Springer-Verlag, Berlin, March 1989.

[Hol83] Sören Holmström. *PFL: A Functional Language for Parallel Programming, and its Implementation.* Programming Methodology Group 7, University of Göteborg and Chalmers University of Technology, September 1983.

[INM84] *occam Programming Manual.* 1984. Prentice-Hall International Series in Computer Science, C.A.R. Hoare (Series Editor).

[Kah74] Gilles Kahn. *The Semantics of a Simple Language for Parallel Programming.* In *Proceedings of the IFIP Conference*, pages 471–475, IFIP, 1974.

[Lar86] Kim G. Larsen. *Context-Dependent Bisimulation between Processes.* PhD thesis, University of Edinburgh, 1986.

[Mei89] Silvio Meira. *Processes and Functions.* In J. Diaz and F. Orejas, editors, *LNCS 352 : TAPSOFT '89*, pages 286–297, Springer-Verlag, Berlin, March 1989.

[Mil80] Robin Milner. *A Calculus of Communicating Systems.* Volume 92 of *Lecture Notes in Computer Science*, Springer-Verlag, 1980.

[Mil83] Robin Milner. *Calculi for Synchrony and Asynchrony. Theoretical Computer Science*, 25:267–310, 1983.

[Mil88] Robin Milner. *Operational and Algebraic Semantics of Concurrent Processes.* Technical Report ECS-LFCS-88-46, University of Edinburgh, 1988.

[MPW89] Robin Milner, Joachim Parrow, and David Walker. *A Calculus of Mobile Processes.* Technical Report ECS-LFCS-89-86, LFCS, Department of Computer Science, University of Edinburgh, June 1989. Also published as CSR-303-89.

[Nie89] Flemming Nielson. *The Typed λ-Calculus with First-Class Processes.* In *LNCS 366: Proceedings of PARLE 89*, pages 357–373, Springer-Verlag, Berlin, 1989.

[Par81] D. Park. *Concurrency and Automata on Infinite Sequences.* Volume LNCS 104, Springer-Verlag, Berlin, 1981.

[Plo75] G.D. Plotkin. *Call by name, call by value, and the λ calculus. Theoretical Computer Science*, 1:125–159, 1975.

[Plo81] G.D. Plotkin. *A Structural Approach to Operational Semantics.* Technical Report DAIMI FN-19, Aarhus University, September 1981.

[Rep88] J.H. Reppy. *Synchronous Operations as First-class Values.* In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation,* pages 250–259, ACM SIGPLAN, June 1988.

[Tho89] Bent Thomsen. *A Calculus of Higher Order Communicating Systems.* In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, January 1989.*, 1989.

[Tur87] David Turner. *Functional Programming and Communicating Processes.* In de Bakker, Nijman, and Treleaven, editors, *LNCS 259: PARLE Parallel Architectures and Languages Europe,* pages 54–74, Springer-Verlag, Berlin, June 1987.

A Free Variables and Substitution

Notation : (FUNCTION RESTRICTION) $f\backslash L$ is a finite domain function obtained from function f and domain L, by restricting f to the domain $dom(f) - L$. That is, $dom(f\backslash L) = dom(f) - L$ and $(f\backslash L)(x) = f(x)$ if $x \notin L$ ■

Definition : (FREE VARIABLES) The function FV yields the set of variables that appear free in a given construct. ■

Definition : (CHANNEL-VALUED CONSTANTS IN A CONSTRUCT) The function CV yields the set of channel-valued constants that appear in a given construct. ■

Definition : (SUBSTITUTION) A substitution is a finite domain function that maps identifiers in $\mathcal{I}$ to values in Val. ρ is a typical substitution. If the domain of ρ, written $dom(\rho)$ is empty, then the substitution is the identity or null substitution. The application of substitutions on syntactic terms in the language is defined inductively, as given in the following table. ■

Construct "X"	$FV(X)$	$CV(X)$	Substitution Instance "$C\rho$"
x	$\{x\}$	$\emptyset$	$\begin{cases} \rho(x) & \text{if } x \in dom(\rho) \\ x & \text{otherwise} \end{cases}$
c	$\emptyset$	$\begin{cases} \{c\} & \text{if } c \in S \\ \emptyset & \text{otherwise} \end{cases}$	c
$\lambda x.e$	$FV(e) - \{x\}$	$CV(e)$	$\lambda x.(e\rho')$ where $\rho' = \rho\backslash\{x\}$.
$e_1\ e_2$	$FV(e_1) \cup FV(e_2)$	$CV(e_1) \cup CV(e_2)$	$(e_1\rho)\ (e_2\rho)$
$\mathtt{channel}(t)$	$\emptyset$	$\emptyset$	$\mathtt{channel}(t)$
$e_1!e_2$	$FV(e_1) \cup FV(e_2)$	$CV(e_1) \cup CV(e_2)$	$(e_1\rho)\ !\ (e_2\rho)$
$e?$	$FV(e)$	$CV(e)$	$(e\rho)\ ?$
$\mathtt{code}(B)$	$FV(B)$	$CV(B)$	$\mathtt{code}(B\rho)$
$\mathtt{spawn}(B)$	$FV(B)$	$CV(B)$	$\mathtt{spawn}(B\rho)$
$\mathtt{terminate}$	$\emptyset$	$\emptyset$	$\mathtt{terminate}$
$\mathtt{activate}\ e$	$FV(e)$	$CV(e)$	$\mathtt{activate}\ (e\rho)$
$B_1 \parallel B_2$	$FV(B_1) \cup FV(B_2)$	$CV(B_1) \cup CV(B_2)$	$(B_1\rho) \parallel (B_2\rho)$
$B_1\ \%\ B_2$	$FV(B_1) \cup FV(B_2)$	$CV(B_1) \cup CV(B_2)$	$(B_1\rho)\ \%\ (B_2\rho)$

Definition : (CHANNEL COMPONENT OF A LABEL) CC is defined to be the set containing the communication channel of a label $l \in Obs$, should it exist. We also extend the definition of CV to labels in Obs. ■

Label l	$CV(l)$	$CC(l)$
$k(v)$	$\{k\} \cup CV(v)$	$\{k\}$
$\overline{k}(v)$	$\{k\} \cup CV(v)$	$\{k\}$
ϵ	$\emptyset$	$\emptyset$