Concurrency Combinators for Declarative Synchronization

Paweł T. Wojciechowski

Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
`Pawel.Wojciechowski@epfl.ch`

Abstract. Developing computer systems that are both *concurrent* and *evolving* is challenging. To guarantee consistent access to resources by concurrent software components, some synchronization is required. A synchronization logic, or policy, is at present entangled in the component code. Adding a new component or modifying existing components, which may require a change of the (global) synchronization policy, is therefore subjected to an extensive inspection of the complete code. We propose a calculus of *concurrency combinators* that allows a program code and its synchronization policy to be expressed separately; the policies include true parallelism, sequentiality, and isolation-only transactions. The calculus is equipped with an operational semantics and a type system. The type system is used to verify if a synchronization policy declared using combinators can be satisfied by program execution.

1 Introduction

Our motivating example of evolving systems are web services [26], i.e. software components which process html or XML data documents received from the network. Due to efficiency and networking reasons, various data fragments may be processed by concurrent services, with possible dependencies on other services. To provide data consistency, transactions or barrier synchronization are required. Unfortunately, a given composition of service components may often change in order to adapt to a new environment or changing requirements, thus making programming of such components a difficult task. We therefore study *declarative synchronization*, which assumes separation of an object's functional behaviour and the synchronization constraints imposed on it. Such approach enables to modify and customize synchronization policies without changing the code of service components, thus making programming easier and less error-prone.

While some work on such separation of concerns exists (see [11, 8, 23, 19] among others), there are few efforts that formalized languages with declarative synchronization in mind. There are also many open questions: How to know when it is safe to spawn a new thread or call a method, so that the synchronization policy is not invalidated? Or, conversely, how can we build programs that are synchronization safe by construction? How should synchronization itself be implemented? What new language features are required?

In this paper, we present a small language with the goal of understanding the underlying foundations of declarative synchronization. The language allows the programmer to group expressions (methods, objects, etc.) into *services*, and to express an arbitrary synchronization policy that will constrain the concurrent execution of services at runtime. Our basic synchronization policies include true parallelism, sequentiality, and the combination of these two policies called *isolation*, which is analogous to the isolation non-interference property known from multithreaded transactions [9, 3]. Isolated services can be assumed to execute serially, without interleaved steps of other services, which significantly simplifies both programming and reasoning about program correctness.

The synchronization policy is expressed abstractly using *concurrency combinators*, i.e. compositional policy operators which take as arguments service names. They introduce a new problem, however. Separation of synchronization and the functional code of a program gives a way of expressing synchronization schemes which may not be satisfied by the program's execution, leading to problems such as deadlock. Any given program can only be executed for some range of synchronization schemes; the synchronization policies must be matched accordingly. In this paper, we propose a type system that can verify if the matching of a policy and program is correct. Typable programs are guaranteed to satisfy the declared synchronization policy and make progress.

The paper is organized as follows. §2 gives a small example. §3 introduces our calculus, describing its syntax and types. §4 defines the operational semantics. §5 describes the type system and main results. §6 contains related work, and §7 concludes.

2 Example

We begin with an example. Below is a term of the *concurrency combinators* calculus (or the CK-calculus, in short) showing how a synchronization policy can be expressed. The program expression is defined using the call-by-value λ-calculus [22], extended with service binders and typing.

$$
\begin{array}{l}
D = \texttt{rule}\ B\ \texttt{foll}\ A \\
\texttt{rule}\ D\,;\,C \\
\texttt{let}\ r = \texttt{ref}\ 0\ \texttt{in} \\
\texttt{let}\ f_A = \lambda^{\{A\},\{A\}} x : t.\, A\ \texttt{\#}\ (r := x;\ ())\ \texttt{in} \\
\texttt{let}\ f_B = \lambda^{\{B\},\{B,A\}} y : t.\, B\ \texttt{\#}\ (\texttt{fork}\ (f_A\ \ y);\ !r)\ \texttt{in} \\
\texttt{let}\ f_C = \lambda^{\{C\},\{C\}} z : t.\, C\ \texttt{\#}\ z \\
\texttt{in} \\
(f_C\ \ (f_B\ \ 1)) \quad \textit{(* this returns } 0\textit{, never } 1\ \textit{*)}
\end{array}
$$

The program declares a synchronization rule (B `foll` A) ; C, which orders services A, B and C to be executed, so that: (1) A runs simultaneously with B but the overall result of their concurrent execution must be equivalent to an (ideal) serial execution in which A commences after B has completed; (2) service C can commence only after *both* A and B have completed.

The service names A, B and C are bound (using `#`) to functions, respectively f_A, f_B, and f_C. The function f_A updates a shared reference cell r (which initially contains 0), function f_B spawns a new thread (using `fork`) for updating the cell (by invoking f_A) in parallel with reading the current content of r. The last function, f_C, only returns its argument. Functions are decorated with a pair of service names that can be bound by a thread calling a function until the function returns, and service names that (also) can be bound by threads spawned as the effect of calling the function, e.g. f_B is decorated with $(\{B\}, \{B, A\})$.

The main expression of the program (following function declarations) calls function f_C with its argument being the result of f_B's call.

The synchronization rule defines a synchronization policy, which constrains creation and interleaving of threads by the runtime system, so that only program executions that match the policy are possible. For instance, our program with the synchronization rule removed, can return either 0 or 1, depending on how the thread spawned inside function f_B is interleaved with the main program thread. The synchronization rule ensures, however, that *each* execution of this program returns 0, never 1; i.e. assignment $r := x$ by service A (where $x = 1$) must block until service B has read the content of r with $!r$.

One may notice that the nested function calls in the main expression (see the last line of the program) impose some ordering on services. This means that some synchronization rules may not be satisfied. A synchronization rule can be *satisfied* by a program if there exists a deadlock-free execution of the program that satisfies the rule. We go back to this issue in §5, where we define a type system which is able to verify combinator satisfiability statically.

3 Language

3.1 Syntax

The syntax of the CK calculus is in Fig. 1. The main syntactic categories are concurrency combinators and expressions. We also assume the existence of reference cells, with the standard semantics of dereference ($!r$) and assignment ($:=$) operations.

Services. *Services* are expressions that are bound to service names, ranged over by A, B, C. We assume that service expressions are themselves deadlock-free. A *composite service* is a collection of services, called *subservices*, whose (possibly) concurrent execution is controlled by a collection of combinator declarations (explained below). A service *completes* if it returns a value. A composite service completes if all its subservices return a value. Below we confuse services and composite services unless stated otherwise.

Concurrency Combinators. *Concurrency combinators* are special functions with no free variables, ranged over by a, b, which are used to define fine-grain concurrency control over the execution of concurrent services. Formal parameters of combinators are service names. We have four *basic combinators*: $A \parallel B$, $A\,;\,B$, A `isol` B, and A `foll` B, where A and B are service names:

Variables	$x, y \in \mathit{Var}$	
Service names	$A, B \in \mathit{Mvar}$	
Packages	$p \in 2^{\mathit{Mvar}} \times 2^{\mathit{Mvar}}$	
Combinators	a, b	$::= A \mid a \parallel b \mid a\,;\,b \mid a\ \texttt{isol}\ b \mid a\ \texttt{foll}\ b$
Types	t	$::= \texttt{Unit} \mid t \to^{p} t'$
Values	$v, w \in \mathit{Val}$	$::= () \mid \lambda^{p} x : t.\, e$
Declarations	K	$::= \texttt{rule}\ a \mid A = \texttt{rule}\ a \mid K\ K$
Expressions	$e \in \mathit{Exp}$	$::= x \mid v \mid e\ \ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{fork}\ e \mid A\ \texttt{\#}\ e$
Program terms	P	$::= K\ e$

We work up to alpha-conversion of expressions throughout, with x binding in e in expressions $\lambda^{p} x : t.\, e$ and $\texttt{let}\, x = e'\ \texttt{in}\ e$.

Fig. 1. The λ-calculus with concurrency combinators

- the "parallel" combinator $A \parallel B$ declares services bound to formal parameters of the combinator to be executed by interleaved threads;
- the "sequence" combinator $A\,;\,B$ declares services bound to A and B to be executed sequentially, i.e. B can be executed by a thread only if all other concurrent threads executing A (if any) have terminated;
- the "isolation" combinator $A\ \texttt{isol}\ B$ allows threads of services bound to A and B to be interleaved, with fine-grain parallelism, but their concurrent execution must satisfy the isolation property, i.e. be equivalent to a serial execution of A and B (as defined in §4);
- the "followed-by" combinator $A\ \texttt{foll}\ B$ is like $A\ \texttt{isol}\ B$, with a constraint that the comparable (ideal) serial run has to be "A followed by B".

If services A and B are not concurrent but they are part of a singlethreaded term, then the combinator $A\ \texttt{isol}\ B$ is always satisfied, and combinators $A\,;\,B$ and $A\ \texttt{foll}\ B$ require only that the execution of A must commence before B.

Combinator Declarations. Concurrency combinators can be declared using *synchronization rules*, ranged over by K, which are terms of the form $\texttt{rule}\ a$ and $A = \texttt{rule}\ a$. Complex combinators $A_0\ op_0\ \ldots\ op_{n-1}\ A_n$ are equivalent to a conjunction of n binary combinators $A_i\ op_i\ A_{i+1}$, where $i = 0..n{-}1$ and op_i is one of the combinator names, i.e. $\parallel$ ("parallel"), ; ("sequence"), isol ("isolation") and foll ("followed-by"). For instance, $\texttt{rule}\ A\,;\,B\ \texttt{isol}\ C$ declares a combinator which ensures that service B can commence only after A has completed, and the execution of B is isolated from any concurrent execution of service C.

Service Composition. Combinators can be arguments of other combinators. For instance, a combinator declaration $A = \texttt{rule}\ a$ both declares a combinator a and also defines a fresh service name A, which binds a combinator a. The name A can then be used in the declaration of other combinators of composite services. (We adopt such syntax since it is convenient to express the semantics rules; the concrete syntax would use parentheses instead.)

Types. Types include the base type `Unit` of unit expressions and the type $t \rightarrow^p t'$ of functions. It would be routine to add subtyping on types to the calculus definition. To verify combinator satisfiability in function abstractions, the type of functions is decorated with a *service package* $p = (p_r, p_s)$, where p_r is a set of all service names which can be bound by a thread calling a function until the function returns, and p_s is the same as p_r but also includes service names bound by any threads spawned as the effect of calling the function. We assume that a programmer can provide information on services bound by functions explicitly, and leave type inference as an open problem.

Values. A value is either an empty value () of type `Unit`, or function abstraction $\lambda^p x : t.\, e$ (decorated with service package p). All values in the CK-calculus are first-class programming objects, i.e. they can be passed as arguments to functions and returned as results and stored in data structures.

Expressions. Basic expressions are mostly standard, including variables, values, function applications, and `let`-binders. The CK language allows multithreaded programs by including the expression `fork` e, which spawns a new thread for the evaluation of expression e. This evaluation is performed only for its effect; the result of e is never used. To bind a service name to an expression, we provide *service binders* of the form A `#` e, which allow an expression e to be bound to a (non-composite) service name A. In programs, the service expression A `#` e is usually a body of a function. We use syntactic sugar $e_1; e_2$ for `let` $x = e_1$ `in` e_2 (for some x, where x is fresh).

Programs. A *program* is a pair (SP, $K\ e$) of a synchronization policy SP and the program expression $K\ e$. A *synchronization policy* (or a *policy* in short) is a set of binary combinators $A\ op\ B$ and bindings of the form $A = B_0\ \ op_0\ ...\ op_{n-1}\ \ B_n$, that are extracted from declarations K, where op_i are combinator names, describing operations $\|$, ;, `isol` and `foll`.

We assume that every policy SP is *logically consistent*, i.e. (1) if A and B in SP are related by operation *op* (possibly via names of composite services whose A or B are subservices) then neither A and B nor B and A are related by $op' \neq op$, and (2) if A and B are related by an asymmetric operation (i.e. `foll` or ;) then B and A are not related. For instance, a policy $C = A \parallel B$ and $C\,;\,A$ for some service names A and B, is not consistent (both conditions are not satisfied). We leave consistency verification as an open problem.

Given the above requirement of consistency, a policy can be easily identified with a sequence of combinator declarations. For instance, in case of our program in §1, SP $= \{D = B$ `foll` $A,\ D;C,\ B$ `foll` $A\}$.

4 Operational Semantics

We specify the operational semantics using the rules defined in Fig. 2 and 3. A state S is a collection of expressions, which are organized as a sequence $T_0, ..., T_n$, where each T_i in the sequence represents a *thread*. We use T, T' (with comma) to denote an unconstrained execution of threads T and T', and $T; T'$ (with

State Space:

$$
\begin{array}{rcl}
S \in \mathit{State} & = & \mathit{ThreadSeq} \\
T, c \in \mathit{ThreadSeq} & ::= & f \mid T, T' \mid T; T' \mid (T) \\
f \in \mathit{Exp}_{ext} & ::= & x \mid v \mid f\ \ e \mid v\ \ f \mid \texttt{let}\ x = f\ \texttt{in}\ e \mid \texttt{let}\ x = v\ \texttt{in}\ f \mid \texttt{fork}\ e \mid \\
& & A\ \texttt{\#}\ e \mid A\{\,c\,\}
\end{array}
$$

Evaluation and Service Contexts:

$$
\begin{array}{ll}
\mathcal{E} = [\,] \mid \mathcal{E}\ \ e \mid v\ \ \mathcal{E} \mid \texttt{let}\ x = \mathcal{E}\ \texttt{in}\ e \mid A\{\,\mathcal{E}\,\} \mid \mathcal{E}, T \mid T, \mathcal{E} \mid \mathcal{E}; T \mid v; \mathcal{E} \mid (\mathcal{E}) & \\
\mathcal{C} = [\,] \mid \overline{A\ op}\ \mathcal{C}\ \overline{op'\ A'} & op \in \{\texttt{isol}, \texttt{foll}, \|, ;\} \\
A_{\mathcal{E}\,T}\{\,c\,\} = \mathcal{E}[\,A\{\,c\,\}\,]; T & \text{for some } T
\end{array}
$$

Structural Congruence Rules:

$$
\begin{array}{lll}
T, T' \equiv T', T & & \text{(C-Sym)} \\
T \circ () \equiv T & \text{where } \circ \text{ is , or ;} & \text{(C-Nil)} \\
(); T \equiv T & & \text{(C-Seq)} \\
a \parallel b \equiv b \parallel a & & \text{(C-Prl)} \\
a\ \texttt{isol}\ b \equiv b\ \texttt{isol}\ a & & \text{(C-Isol)}
\end{array}
\qquad
\frac{T \longrightarrow T'}{\mathcal{E}[\,T\,] \longrightarrow \mathcal{E}[\,T'\,]}\ \text{(C-Expr)}
$$

Fig. 2. The CK-calculus: Operational semantics

semicolon) to denote that T' can commence only after T has reduced to a value, and (T) for grouping threads. We write $T \circ T'$ to mean either T, T' or $T; T'$.

The expressions f are written in the calculus presented in §3.1, extended with a new construct $A\{\,c\,\}$, which denotes a sequence of threads c that is part of service A. The construct is not part of the language to be used by programmers; it is just to explain semantics.

We define a small-step evaluation relation $e \longrightarrow e'$ read "expression e reduces to expression e' in one step". We also use $\longrightarrow^*$ for a sequence of small-step reductions, and a "meta" relation $\twoheadrightarrow$ (defined below) for many reduction steps with the isolation guarantee. Reductions are defined using evaluation context $\mathcal{E}$ for expressions and threads, and context $\mathcal{C}$ for synchronization policy rules. Context application is denoted by $[]$, as in $\mathcal{E}[\,e\,]$. We write $\overline{A\ op}$ as shorthand for a possibly empty sequence $A\ op \ldots A'\ op'$ (and similarly for $\overline{op\ A}$).

We also use an abbreviation $A_{\mathcal{E}\,T}\{\,c\,\}$ for $\mathcal{E}[\,A\{\,c\,\}\,]; T$ — i.e., "a context $\mathcal{E}$ of service A, followed by a (possibly empty) thread T or a group of threads T that are blocked until A will complete". To lighten semantics rules, we usually omit T and write $A_{\mathcal{E}}\{\,c\,\}$.

Structural congruence rules are defined in Fig. 2. They can be used to rewrite thread expressions and synchronization policy rules whenever needed.

The evaluation of a program $(\mathrm{SP}, K\ e)$ starts in an initial state with a single thread that evaluates the program's expression e. Evaluation then takes place according to the transition rules in Fig. 3. The rules specify the behaviour of the constructs of our calculus. The evaluation terminates once all threads have

Evaluator:

$$eval \subseteq \; Exp \times Val$$
$$eval(e,v) \Leftrightarrow e \longrightarrow^* T_0 \circ ... \circ T_n \;\; \text{and} \;\; T_0 = v, \; T_{j\neq 0} = ()$$

Transition Rules:

$$S + S' \longrightarrow S \;\; \text{or} \;\; S + S' \longrightarrow S' \qquad \text{(R-Choice)}$$

$$\lambda x.\, e \;\; v \longrightarrow \{v/x\}e \qquad \text{(R-App)}$$

$$\texttt{let}\; x = v \;\texttt{in}\; e \longrightarrow \{v/x\}e \qquad \text{(R-Let)}$$

$$A \;\texttt{\#}\; e \longrightarrow A\{\, e \,\} \qquad \text{(R-Bind)}$$

$$A\{\, v \,\} \longrightarrow v \qquad \text{(R-Compl)}$$

$$\mathcal{E}[\, \texttt{fork}\; e \,] \longrightarrow \mathcal{E}[\, () \,], e \qquad \text{(R-Fork)}$$

$$A\{\, c \,\}; T, A\{\, c' \,\}; T' \longrightarrow A\{\, c, c' \,\}; (T, T') \qquad \text{(R-Join)}$$

$$\frac{A = \mathcal{C}[\, B \; op \; \mathcal{C}'[\, C \,]\,] \in \text{SP} \qquad \mathcal{E}''[\, c'' \,] \neq \mathcal{E}'''[\, A\{\, c'' \,\}\,]}{\mathcal{E}''[\, B_{\mathcal{E}}\{\, c \,\} \circ C_{\mathcal{E}'}\{\, c' \,\}\,] \longrightarrow \mathcal{E}''[\, A\{\, B_{\mathcal{E}}\{\, c \,\} \circ C_{\mathcal{E}'}\{\, c' \,\}\,\}\,]} \qquad \text{(R-Fold)}$$

$$\frac{\begin{array}{c} A,\, B \text{ are innermost services of redex} \\ A = B_1 \; op \; ... \; op' \; B_n \in \text{SP} \qquad B \neq B_i \quad i = 1..n \end{array}}{A\{\, T \circ B_{\mathcal{E}}\{\, c \,\}, T' \,\} \longrightarrow A\{\, T, T' \,\} \circ B_{\mathcal{E}}\{\, c \,\}} \qquad \text{(R-Unfold)}$$

$$\frac{A \,;\, B \in \text{SP}}{A_{\mathcal{E}}\{\, c \,\}, B_{\mathcal{E}'}\{\, c' \,\} \longrightarrow A_{\mathcal{E}}\{\, c \,\}; B_{\mathcal{E}'}\{\, c' \,\}} \qquad \text{(R-Seq)}$$

$$\frac{\begin{array}{c} A \;\texttt{foll}\; B \in \text{SP} \\ \mathcal{E}''[\, A_{\mathcal{E}}\{\, c \,\}; B_{\mathcal{E}'}\{\, c' \,\}\,] \longrightarrow^* S \end{array}}{\mathcal{E}''[\, A_{\mathcal{E}}\{\, c \,\}, B_{\mathcal{E}'}\{\, c' \,\}\,] \twoheadrightarrow S} \qquad \text{(R-Foll)}$$

$$\frac{\begin{array}{c} A \;\texttt{isol}\; B \in \text{SP} \\ \mathcal{E}''[\, A_{\mathcal{E}}\{\, c \,\}; B_{\mathcal{E}'}\{\, c' \,\}\,] \longrightarrow^* S \\ \mathcal{E}''[\, B_{\mathcal{E}'}\{\, c' \,\}; A_{\mathcal{E}}\{\, c \,\}\,] \longrightarrow^* S' \end{array}}{\mathcal{E}''[\, A_{\mathcal{E}}\{\, c \,\}, B_{\mathcal{E}'}\{\, c' \,\}\,] \twoheadrightarrow S + S'} \qquad \text{(R-Isol)}$$

To lighten notation, we write $A_{\mathcal{E}}\{\, c \,\}$ instead of $A_{\mathcal{E}\,T}\{\, c \,\}$.
We assume that if SP has $A; B$ or A `foll` B for some A, then evaluation of $B\{\, c \,\}$ (or $B_i\{\, c_i \,\}$ and $i = 1..n$, if $B = B_1 \; op \; ... \; op' \; B_n$) is blocked till rules (R-Seq) or (R-Foll) are applied to A and B, unless no other redex is available.

Fig. 3. The CK-calculus: Operational semantics

been reduced to values, in which case the value v of the initial, first thread T_0 is returned as the program's result. (A typing rule for `fork` will ensure that other values are empty values.) The execution of unconstrained threads can be arbitrarily interleaved. Since different interleavings may produce different results, the evaluator $eval(e, v)$ is therefore a relation, not a partial function. Below we describe the evaluation rules.

Nondeterministic choice (R-Choice) between states S and S', denoted $S + S'$, can lead to either S being evaluated and S' discarded, or opposite.

The next two evaluation rules are the standard rules of a call-by-value λ-calculus [22]. Function application $\lambda x.\, e \;\; v$ in rule (R-App) reduces to the function's body e in which a formal argument x is replaced with the actual argument v. The (R-Let) rule reduces $\texttt{let}\ x = v\ \texttt{in}\ e$ to the expression in which variable x is replaced by value v in e. We write $\{v/x\}e$ to denote the capture-free substitution of v for x in the expression e.

Service binder $A \,\texttt{\#}\, e$ in rule (R-Bind) marks an expression e with the service name A bound to e; it reduces to the expression $A\{\, e \,\}$. The marking information will allow concurrency control rules (described below) to identify expressions that are part of a given service, and apply to them all relevant synchronization rules while evaluating the expressions.

The mark $A\{\, e \,\}$ will be erased when expression e evaluates to a value v (see (R-Compl)). Then, we say that service A has *completed*.

Evaluation of expression `fork` e creates a new thread which evaluates e (see (R-Fork)). A value returned by expression e will be discarded. Note that threads spawned by a (non-composite) service A will not be part of service A, unless the expression spawned by the new thread is explicitly bound to A.

4.1 Concurrency Combinators

The rules at the bottom half of Fig. 3, beginning from (R-Join), define the semantics of concurrency control. Programs evaluated using the rules must be first checked if the rules can be actually applied for a given synchronization policy. In §5, we present a type system that can verify this condition.

The first rule, (R-Join), groups two concurrent subexpressions of the same service. The rule (R-Fold) encloses two concurrent services that are part of a composite service A with the name A. The rule (R-Unfold) removes service B (together with any threads blocked on B) outside the scope of a composite service A whose B is *not* part of. Note that abbreviations $A_{\mathcal{E}}\{\, c \,\}$ and $A_{\mathcal{E}'}\{\, c' \,\}$ allow contexts $\mathcal{E}$ and $\mathcal{E}'$ to be multithreaded, if needed by reduced expressions.

The rule (R-Seq) blocks a thread (or threads) of service B until service A would complete. To explain other rules, we need to introduce a few definitions.

Isolation Property. By *concurrent execution*, we mean a sequence of small-step reductions in which the reduction steps can be taken by threads with possible interleaving. Two (possibly multithreaded) services are executed *serially* if one service commences after another one has completed.

A *result* of evaluating a service expression bound to A is any state S, that does not have a context $\mathcal{E}[\, A\{\, ..\, \}\,]$. Note that states subsume the content of reference cells, represented with stores (the details are omitted in Fig. 3 as our main focus in this paper is on verification of combinators). An *effect* is any change to the content of stores.

We define *isolation* to mean that the effects of one service are not visible to other services executing concurrently—from the perspective of a service, it appears that services execute serially rather than in parallel.

The operational semantics of combinators A `isol` B and A `foll` B, is captured using rules (R-Isol) and (R-Foll). The rules define the "isolated evaluation" relation ($\twoheadrightarrow$). They specify that the actual term which contains services A and B (in the conclusion of each rule) should be evaluated by the single-step reduction ($\longrightarrow$) using all evaluation rules but (R-Isol) and (R-Foll). However, the order of applying the rules must be now constrained, so that *any* result S or S' of such concurrent evaluation of the term, could be also obtained by evaluating a less concurrent term – given in the premises of rules (R-Isol) and (R-Foll) – in which services A and B are executed serially.

The combinator A `foll` B restricts the (ideal) serial execution to "A followed by B", while combinator A `isol` B does not specify the order.

We assume that if SP has $A; B$ or A `foll` B for some A, then evaluation of $B\{\, c\, \}$ (or $B_i\{\, c_i\, \}$ and $i = 1..n$, if $B = B_1\ op\ ...\ op'\ B_n$) is blocked till rules (R-Seq) or (R-Foll) are applied to A and B, unless no other redex is available.

Implementation of Isolation. An implementation of concurrency combinators should schedule threads of well-typed expressions so that the SP policy is effectuated. The most interesting case is the runtime support for isolation of composite services that are themselves concurrent. In our previous work [27], we have designed several fine-grain concurrency control algorithms that provide a deadlock-free implementation of such type of scheduling. They have been used to implement SAMOA—a library package in Java for developing modular network protocols, in which network and application messages can be processed by isolated, possibly multithreaded tasks. Programmers can spawn tasks using the `isolated` e construct, where e is the task's code. We have used our package to implement example protocols for server replication. The `isolated` construct made programming of protocols easier [27], especially those with many threads. The isolation property ensures that each message is always processed by a new, fresh task, using a consistent set of session and message-specific data.

5 Typing for Combinator Satisfiability

In order to implement concurrency combinators, one must decide if the combinators should be considered as powerful typing, or as directives for the runtime system to schedule operations, or maybe both, and if both, then to which degree one could depend on static analysis, and what must be supported by the runtime tools. In this paper we claim that programs could and should be verified statically if the declared synchronization rules can be actually satisfied (considering

Judgments:

$\Gamma; b; p \vdash e : t$ — e is a well-typed expression of type t in Γ, bound to service names in $b = (b_r, b_s)$ of service package $p = (p_r, p_s)$

Expression Typing:

$$\frac{x : t \in \Gamma}{\Gamma; b; p \vdash x : t} \quad \text{(T-Var)}$$

$$\frac{\Gamma; b; p \vdash e : t \quad (\Gamma, x : t); b'; p' \vdash e' : t' \quad b \subseteq b' \quad p \subseteq p'}{\Gamma; b'; p' \vdash \mathtt{let}\ x = e\ \mathtt{in}\ e' : t'} \quad \text{(T-Let)}$$

$$\frac{\Gamma, x : t; \emptyset; p \vdash e : t'}{\Gamma; b; p' \vdash \lambda^p x : t.\, e : t \to^p t'} \quad \text{(T-Abs)}$$

$$\frac{}{\Gamma; b; p \vdash () : \mathtt{Unit}} \quad \text{(T-Unit)}$$

$$\frac{\Gamma; (\emptyset, b_s); (\emptyset, p_s) \vdash e : t}{\Gamma; b; p \vdash \mathtt{fork}\ e : \mathtt{Unit}} \quad \text{(T-Fork)}$$

$$\frac{\begin{array}{c}\Gamma; b; p' \vdash e : t' \to^p t \\ \Gamma; b'; p'' \vdash e' : t' \quad b \subseteq b' \quad p' \subseteq p'' \\ \forall A \in p_r\ \nexists B \in b'_r.\ (A, B)\ \text{prl} \\ \forall A \in p_s\ \nexists B \in b'_s.\ (A, B)\ \text{seq}\end{array}}{\Gamma; b' \cup p; p'' \cup p \vdash e\ e' : t} \quad \text{(T-App)}$$

$$\frac{\begin{array}{c}\Gamma; b \cup \{A\}; p \vdash e : t \\ A \in p_r \quad A \in p_s \\ \nexists B \in b_r.\ (A, B)\ \text{prl} \\ \nexists B \in b_s.\ (A, B)\ \text{seq}\end{array}}{\Gamma; b \cup \{A\}; p \vdash A\ \mathtt{\#}\ e : t} \quad \text{(T-Bind)}$$

Auxiliary Definitions:

$$\frac{A \parallel B \in \text{CS or } B \parallel A \in \text{CS}}{(A, B)\ \ \text{prl}} \quad \text{(T-Prl)}$$

$$\frac{\begin{array}{c}A; B \in \text{CS} \\ \text{or } A\ \mathtt{foll}\ B \in \text{CS}\end{array}}{(A, B)\ \ \text{seq}} \quad \text{(T-Seq)}$$

$x \cup p = (x_r \cup p, x_s \cup p)$ $\qquad$ $x \subseteq x' \equiv x_r \subseteq x'_r$ and $x_s \subseteq x'_s$ $\qquad$ where $x = b$ or $x = p$

Fig. 4. The CK-calculus: Typing expressions

any implementation of combinators). Programs that cannot satisfy a given rule simply would not compile.

Consider our example program in §2, with a new policy $D = B\ \mathtt{foll}\ A$ and $C\,;\,D$. Unfortunately, *all* possible executions of the program can no longer satisfy the latter rule. The new policy requires that service C, which binds function f_C, must return before services A and B can commence. With such a policy, the program is not able to begin its execution since the function f_C is expected to be called with an argument returned by the call of f_B. However, according to the new policy, function f_B cannot be called before f_C returns. So, we have a deadlock situation. In this section, we define a type system that can statically verify combinator satisfiability, and so detect the possibility of deadlock.

Satisfiability of Combinators. The semantics can be used to formalize the notion of *combinator satisfiability*, as follows. A thread T *binds* a service name A if there exists some evaluation context $\mathcal{E}$ such that $T = \mathcal{E}[\,A\{\,f\,\}\,]$ for some expression f. A state S does *not* satisfy combinator $A \parallel B$ if its thread sequence contains a thread that binds both service names A and B. A state S does not satisfy combinator $A\,;\,B$ and also combinator A `foll` B, if its thread sequence contains either a term $\mathcal{E}[\,B\{\,f\,\}\,]$ such that $f = \mathcal{E}'[\,A \;\texttt{\#}\; e\,]$ (possibly $f = \mathcal{E}'[\,\texttt{fork}\;\mathcal{E}''[\,A \;\texttt{\#}\; e\,]\,]$), or a (single- or multithreaded) term $\mathcal{E}[\,B\{\,c\,\}\,];T$ and $T = \mathcal{E}'[\,A\{\,c'\,\}\,]$, for some contexts $\mathcal{E}$, $\mathcal{E}'$ and $\mathcal{E}''$ and expressions f, e, c and c'. In all other cases, the above combinators are *satisfied*. Finally, combinator A `isol` B can be satisfied at runtime by all execution states. This is because any singlethreaded evaluation of services A and B ensures the isolation property trivially. Otherwise, if A and B are evaluated by different threads, then rule (R-Isol) is applied to scheduling threads accordingly.

An execution run $S \longrightarrow^* S'$ does not satisfy a combinator a if it may yield a state that does not satisfy a, i.e. if there exists a state S'' in the run (including S and S') such that S'' does not satisfy a. Otherwise, we say that the run $S \longrightarrow^* S'$ *satisfies* combinator a.

Typing for Combinator Satisfiability. We define the type system using one judgment for expressions. The judgment and the static typing rules for reasoning about the judgment are given in Fig. 4. The typing judgment has the form $\Gamma; b; p \vdash e : t$, read "expression e has type t in environment Γ with bound service names b of service package p", where an environment Γ is a finite mapping from free variables to types. A *package* $p = (p_r, p_s)$ is defined by all service names which may be bound while evaluating expression e, either by the current thread only (p_r) or by all threads evaluating e (p_s); if e is single-threaded then $p_r = p_s$.

Our intend is that, given a policy SP if the judgment $\Gamma; b; p \vdash e : t$ holds, then expression e can satisfy all concurrency combinators in SP, and yields values of type t, provided the current thread binds services described in b, it may bind at most services described in p, and the free variables of e are given bindings consistent with the typing environment Γ.

The core parts of typing rules for expressions are fairly straightforward and typical for the λ-calculus with threads evaluated only for their side-effects. The only unusual rule is (T-Bind); it type checks service binders of the form $A \;\texttt{\#}\; e$, and requires the type of the whole expression to be e's type. To support modular design of services, service binders are typable only if they are inside a function.

The main novelty is verifying if expressions satisfy combinators. For this, we check if expressions do not invalidate any constraints imposed by the policy rules. A *set of constraints* CS is constructed recursively from a set of binary combinators copied from the policy SP (we assume a fixed policy for each program). For each binding rule $C = A_1\ op\ ...\ op'\ A_n$ in SP, replace every combinator c of the form $X\ op\ Y$ where $X = C$ or $Y = C$, by n constraints c_i, such that c_i is exactly the same as c but the name C (on a given position) is replaced by A_i, e.g. in case of our example program in §2, $\mathrm{CS} = \{B \;\texttt{foll}\; A,\ B;C,\ A;C\}$.

During typechecking, an expression is evaluated for a given constraint set CS in the context of a package p, and a pair $b = (b_r, b_s)$ of service names b_r that have been bound by the current thread explicitly, using `#`, and service names b_s that are the same as in b_r but also include names inherited by the current thread at spawning time, from a set b_s of the parent thread.

For instance, consider typing the operation of binding a new service name A. The (T-Bind) rule checks if no constraint among those that define relations (A, B) prl and (A, B) seq (see in the bottom of Fig. 4) appeared in CS, where B is any service bound so far by the current thread (i.e. *before* A is bound). If at least one such constraint is in CS, then the program is not typable since the constraint cannot be satisfied by any execution of the program. This is because either A and B are bound in a wrong order (thus violating combinators $A; B$ and A `foll` B) or they are bound by the same thread (thus disabling true parallelism and so violating combinator $A \parallel B$).

A package $p = (p_r, p_s)$ decorates a function type and definition, representing all services that may be bound while evaluating the function by the current thread T only (p_r) and by T and also any other threads that are spawned as the effect of calling the function (p_s). The rule (T-App) checks if relations (A, B) prl and (A, B) seq do not hold, where A is any service in the package implemented by a function, and B is any service bound by a thread calling the function.

The rule (T-Fork) requires the type of the whole expression to be `Unit`; this is correct since threads are evaluated only for their side effects. Note that the `fork`-ed expression is evaluated with b_r and p_r nulled since verification of the $A \parallel B$ combinator requires that any spawned threads do not inherit service bindings from their parent thread (as we only check if A and B are not single-threaded).

The type system could be extended with reachability analysis of conditional branches and dead code elimination. Otherwise, some declared policies would be rejected, even if all program executions may satisfy them.

Well-typed Programs satisfy Combinators. The fundamental property of the type system is that well-typed programs satisfy the declared synchronization policy, expressed using concurrency combinators. The first component of the proof of this property is a type preservation result stating that typing is preserved during evaluation. To prove this result, we extend typing judgments from expressions in *Exp* to states in *State* as shown in Fig. 5. The judgment $\vdash S : t$ says that S is a well-typed state yielding values of type t.

Lemma 1 (Type Preservation). *If $\Gamma; b; p \vdash S : t$ and $S \longrightarrow S'$, then $\Gamma; b; p \vdash S' : t$.*

Lemma 2 states that a program typable for some synchronization policy SP is reducible to states that satisfy all combinators in SP.

Lemma 2 (Combinator Preservation). *Suppose $\Gamma; \emptyset; \emptyset \vdash S : t$ for some synchronization policy* SP*. If $S \longrightarrow^* S'$, then run $S \longrightarrow^* S'$ satisfies all combinators in* SP *up to state S'.*

Type preservation and combinator preservation "up to a state" ensure that if we start with a typable expression for some policy SP, then we cannot reach an untypable expression through any sequence of reductions, and the reduced expression satisfies combinators in SP. This by itself, however, does not yield type soundness. Lemma 3 states that evaluation of a typable expression cannot get *stuck*, i.e. either the expression is a value or there is some reduction defined.

Judgments:

$\vdash S : t$ S is a well-typed state of type t

Rules:

$$\frac{|T| > 0 \quad \Gamma; \emptyset; \emptyset \vdash T_i : t_i \quad \text{for all } \ i < |T|}{\vdash T : t_0} \text{(T-State)} \qquad \frac{\Gamma; b; p \vdash f_i : t_i \quad \Gamma; b'; p' \vdash f'_j : t_j \quad i < j}{\Gamma; b; p \vdash f_i \circ f'_j : t_i} \text{(T-Thread)}$$

$$\frac{\vdash S : t_0 \quad \vdash S' : t_0}{\vdash S + S' : t_0} \text{(T-Choice)} \qquad \frac{\Gamma; b; p \vdash c : t}{\Gamma; b; p \vdash A\{\, c \,\} : t} \text{(T-InService)}$$

Fig. 5. Additional judgments and rules for typing states.

Lemma 3 (Progress). *Suppose S is a closed, well-typed state (that is, $\vdash S : t$ for some t and policy* SP*). Then either S is a value or else, there is some state S' with $S \longrightarrow S'$.*

We conclude that for a given policy SP, well-typed programs satisfy combinators in SP. An expression e is a *well-typed program* if it is closed and it has a type t in the empty type environment, written $\vdash e : t$.

Theorem 1 (Combinator Satisfiability). *Given a policy* SP*, if $\vdash e : t$, then all runs $e \longrightarrow^* v_0$, where v_0 is some value, satisfy combinators in* SP*.*

6 Related Work

There have been recently many proposals of concurrent languages with novel synchronization primitives, e.g. Polyphonic C# [2] and JoCaml [5], which are based on the join-pattern abstraction [7]; and Concurrent Haskell [14], Concurrent ML [20], Pict [21] and Nomadic Pict [24, 28] which have synchronization constructs based on channel abstractions. They enable to encode complex concurrency control more easily than when using standard constructs, such as monitors and locks.

Flow Java [4] extends Java with *single assignment variables*, which allow programmers to defer binding of objects to these variables. Threads accessing an unbound variable are blocked, e.g. the method call `c.m()` will block until `c` has been bound to an object (by other thread). This mechanism can be used to implement barrier synchronization in concurrent programs.

The above work is orthogonal to the goals of this paper. We are primarily focused on a declarative way of encoding and verifying synchronization through separation of concerns (see [11, 16, 8, 23, 18, 19] among others), with higher-level transactional facilities that provide automatic concurrency control. Below we discuss example work in these two areas.

Separation of Concurrency Aspects. The previous work, which set up goals close to our own is by Ren and Agha [23] on separation of an object's functional behaviour and the timing constraints imposed on it. They propose an actor-based language for specifying and enforcing at runtime real-time relations between events in a distributed system. Their work builds on the earlier work of Frølund and Agha [8] who developed language support for specifying multi-object coordination, expressed in the form of constraints that restrict invocation of a group of objects.

For a long time, the object-oriented community has been pointing out, under the term *inheritance anomaly* [17], that concurrency control code interwoven with the application code of classes can represent a serious obstacle to class inheritance, even in very simple situations. Milicia and Sassone [18, 19] address the inheritance anomaly problem, and present an extension of Java with a linear temporal logic to express synchronization constraints on method calls. This approach is similar to ours however we are focused on verifying static constraints between code fragments.

The *Aspect Oriented Programming (AOP)* approach is based on separately specifying the various *concerns* of a program and some description of their relationship, and then relying on the AOP tools to *weave* [12] or compose them together into a coherent program. Hürsch and Lopes [11] identify various concerns, including synchronization. Lopes [16] describes a programming language D, which allows thread synchronization to be expressed as a separate concern. More recently, the AOP tools have been proposed for Java, such as AspectJ [15]; they allow aspect modules to be encoded using traditional languages, and weaved at the intermediate level of Java bytecode.

We are not aware of much work on formalizing combinator-like operations. Achermann and Nierstrasz [1] describe Piccola, which allows software components to be composed (although not isolated) using connectors, with rules governing their composition.

Isolation-only Transactions. A number of researchers describe a way to decompose transactions, and provide support of the isolation property in common programming. For instance, Venari/ML [9] implements higher-order functions in ML to express modular transactions, with concurrency control factored out into a separate mechanism that the programmer could use to ensure isolation.

Flanagan and Qadeer's [6] developed a type system for specifying and verifying the atomicity of methods in multithreaded Java programs (the notion of "atomicity" is equivalent to isolation in this paper). The type system is a synthesis of Lipton's theory of left and right movers (for proving properties of parallel programs) and type systems for race detection.

Harris and Fraser [10] have been investigating an extension of Java with atomic code blocks that implement Hoare's conditional critical regions (CCRs). However, both Flanagan and Qadeer's atomic methods and Harris and Fraser's atomic blocks must be sequential, while our isolated (composite) services can themselves be multithreaded.

Black *et al.* [3] defined an equation theory of atomic transaction operators, where an operator corresponds to an individual ACID (Atomicity, Consistency, Isolation, and Durability) property. The operators can be composed, giving different semantics to transactions. The model is however presented abstractly, without being integrated with any language or calculus.

Vitek *et al.* [25] (see also Jagannathan and Vitek [13]) have recently proposed a calculi-based model of (standard) ACID transactions. They formalized the optimistic and two-phase locking concurrency control strategies.

7 Conclusions

Our small, typed calculus may be a useful basis for work on different problems of declarative synchronization. One problem that we have identified in this paper, and solved using a type system, is satisfiability of combinators and scheduling policies. Such combination of static typing with runtime support would be helpful to implement concurrency combinators. It may be also worthwhile to investigate algorithms for inferring the typing annotations.

We have focused on the simplest language that allows us to study the core problem of §1, rather than attempting to produce an industrial-strength language. We think however that analogous work could be carried out for other languages, too. We hope that having abstractions similar to our concurrency combinators in the mainstream programming languages would facilitate the development of concurrent, service-oriented systems, especially those that need to deal with unanticipated evolution.

Acknowledgments. Research supported by the Swiss National Science Foundation under grant number 21-67715.02 and Hasler Stiftung under grant number DICS-1825.

References

1. F. Achermann and O. Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. In M. Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.
2. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. In *Proc. ECOOP '02*, LNCS 2374, June 2002.
3. A. P. Black, V. Cremet, R. Guerraoui, and M. Odersky. An equational theory for transactions. In *Proc. FSTTCS '03 (23rd Conference on Foundations of Software Technology and Theoretical Computer Science)*, Dec. 2003.
4. F. Drejhammar, C. Schulte, P. Brand, and S. Haridi. Flow Java: Declarative concurrency for Java. In *Proc. ICLP '03 (Conf. on Logic Programming)*, 2003.
5. F. L. Fessant and L. Maranget. Compiling join-patterns. In *Proc. HLCL '98 (Workshop on High-Level Concurrent Languages)*, 1998.
6. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proc. PLDI '03 (Conf. on Programming Language Design and Implementation)*, June 2003.
7. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proc. of CONCUR '96*, LNCS 1119, Aug. 1996.

8. S. Frølund and G. Agha. A language framework for multi-object coordination. In *Proc. ECOOP '93*, LNCS 627, July 1993.
9. N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing. Composing first-class transactions. *ACM TOPLAS*, 16(6):1719–1736, Nov. 1994.
10. T. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. OOPSLA '03*, Oct. 2003.
11. W. Hursch and C. Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Feb. 1995.
12. R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *Proc. ECOOP 2003*, LNCS 2743, July 2003.
13. S. Jagannathan and J. Vitek. Optimistic concurrency semantics for transactions in coordination languages. In *Proc. COORDINATION '04*, Feb. 2004.
14. S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. POPL '96 (23rd ACM Symposium on Principles of Programming Languages)*, Jan. 1996.
15. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, Oct. 2001.
16. C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, Dec. 1997 (1998).
17. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.
18. G. Milicia and V. Sassone. Jeeg: A programming language for concurrent objects synchronization. In *Proc. ACM Java Grande/ISCOPE Conference*, Nov. 2002.
19. G. Milicia and V. Sassone. Jeeg: Temporal constraints for the synchronization of concurrent objects. Tech. Report RS-03-6, BRICS, Feb. 2003.
20. P. Panangaden and J. Reppy. The Essence of Concurrent ML. In F. Nielson, editor, *ML with Concurrency: Design, Analysis, Implementation, and Application*, pages 5–29. Springer, 1997.
21. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
22. G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
23. S. Ren and G. A. Agha. RTsynchronizer: Language support for real-time specifications in distributed systems. In *Proc. ACM Workshop on Languages, Compilers, & Tools for Real-Time Systems*, 1995.
24. P. Sewell, P. T. Wojciechowski, and B. C. Pierce. Location-independent communication for mobile agents: A two-level architecture. In *Internet Programming Languages*, LNCS 1686, pages 1–31, 1999.
25. J. Vitek, S. Jagannathan, A. Welc, and A. L. Hosking. A semantic framework for designer transactions. In *Proc. ESOP '04*, Mar./April 2004.
26. W3C. *Web Services Architecture*, 2004. http://www.w3.org/TR/ws-arch/.
27. P. Wojciechowski, O. Rütti, and A. Schiper. SAMOA: A framework for a synchronisation-augmented microprotocol approach. In *Proc. IPDPS 2004 (18th International Parallel and Distributed Processing Symposium)*, Apr. 2004.
28. P. T. Wojciechowski and P. Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency. The Computer Society's Systems Magazine*, 8(2):42–52, April-June 2000.

A Execution trace

Below is the execution trace of the program in §1 (we erased types, skipped combinator and function declarations, and use name r for the reference location).

$$
\begin{array}{rl}
& D = \texttt{rule}\ B\ \texttt{foll}\ A \\
& \texttt{rule}\ D\,;C \\
& \texttt{let}\ r = \texttt{ref}\ 0\ \texttt{in} \\
& \texttt{let}\ f_A = \lambda x.\, A\ \texttt{\#}\ (r := x;\ ())\ \texttt{in} \\
& \texttt{let}\ f_B = \lambda y.\, B\ \texttt{\#}\ (\texttt{fork}\ (f_A\ \ y);\ !r)\ \texttt{in} \\
& \texttt{let}\ f_C = \lambda z.\, C\ \texttt{\#}\ z\ \texttt{in} \\
& (f_C\ \ (f_B\ \ 1)) \\
& \\
\longrightarrow^* & (\lambda z.\, C\ \texttt{\#}\ z \\
& (\lambda y.\, B\ \texttt{\#}\ (\texttt{fork}\ (f_A\ \ y);\ !r)\ \ 1)) \\
\longrightarrow & (\lambda z.\, C\ \texttt{\#}\ z \\
& (\lambda y.\, B\ \texttt{\#}\ (\texttt{fork}\ (\lambda x.\, A\ \texttt{\#}\ (r := x;\ ())\ \ y);\ !r)\ \ 1)) \\
\longrightarrow & (\lambda z.\, C\ \texttt{\#}\ z \\
& B\ \texttt{\#}\ (\texttt{fork}\ (\lambda x.\, A\ \texttt{\#}\ (r := x;\ ())\ \ 1);\ !r)) \\
\longrightarrow & (\lambda z.\, C\ \texttt{\#}\ z \\
& B\{\,\texttt{fork}\ (\lambda x.\, A\ \texttt{\#}\ (r := x;\ ())\ \ 1);\ !r\,\}) \\
\longrightarrow & (\lambda z.\, C\ \texttt{\#}\ z\ \ B\{\,();\ !r\,\}),\ (\lambda x.\, A\ \texttt{\#}\ (r := x;\ ())\ \ 1) \\
\longrightarrow & (\lambda z.\, C\ \texttt{\#}\ z\ \ B\{\,();\ !r\,\}),\ A\ \texttt{\#}\ (r := 1;\ ()) \\
\longrightarrow & (\lambda z.\, C\ \texttt{\#}\ z\ \ B\{\,();\ !r\,\}),\ A\{\,r := 1;\ ()\,\} \\
\longrightarrow & D\{\,(\lambda z.\, C\ \texttt{\#}\ z\ \ B\{\,();\ !r\,\}),\ A\{\,r := 1;\ ()\,\}\,\} \\
\longrightarrow^* & D\{\,(\lambda z.\, C\ \texttt{\#}\ z\ \ 0),\ A\{\,r := 1;\ ()\,\}\,\} \\
\longrightarrow & D\{\,C\ \texttt{\#}\ 0,\ A\{\,r := 1;\ ()\,\}\,\} \\
\longrightarrow & D\{\,C\{\,0\,\},\ A\{\,r := 1;\ ()\,\}\,\} \\
\longrightarrow^* & D\{\,A\{\,r := 1;\ ()\,\}\,\};\ C\{\,0\,\} \\
\longrightarrow^* & D\{\,()\,\};\ C\{\,0\,\} \\
\longrightarrow & (),\ C\{\,0\,\} \\
\longrightarrow & 0
\end{array}
$$