Formalisation of FunLoft

Frédéric Boussinot, Frederic Dabrowski

Formalisation of FunLoft

Frédéric Boussinot*
EMP-CMA/INRIA - Sophia Antipolis
B.P. 93, 06902 Sophia-Antipolis Cedex, France
Frederic.Boussinot@sophia.inria.fr

Frédéric Dabrowski
IRISA - Rennes
Campus de Beaulieu, 35 042 Rennes Cedex, France
Frederic.Dabrowski@irisa.fr

October 29, 2007

Abstract

We formalise a thread-based concurrent language which makes resource control possible. Concurrency is based on a two-level model: threads are executed cooperatively when linked to a scheduler, and unlinked threads and schedulers are executed preemptively, under the control of the OS. We present a type and effect system to enforce a logical separation of the memory which ensures that (1) when running in preemptive mode, threads do not interfere with other threads; (2) threads linked to a scheduler do not interfere with threads linked to another scheduler. Thus, we get a concurrency model in which well-typed programs are free from data-races. The type system also insures that well-typed programs are bounded in memory and in their use of the CPU. Detection of termination of recursive functions and stratification of references in memory are techniques used to get these properties.

1 Introduction

FunLoft is an experimental programming language mainly concerned with three topics: concurrency, resource control, and safety. Concurrency basically means the use of threads. Resources that are intended to be controlled are the memory and the CPU. Safety basically means absence of problematic concurrent accesses to shared data, called data-races. The objective of FunLoft is however larger than to combine concurrency, resource control, and safety: first, the language proposes a particular form of concurrency, intended to improve standard concurrency frameworks based on preemptive threads (Java threads or Pthreads[20]).

*with support from ACI ALIDECS

Second, FunLoft allows users to benefit from real parallelism (in particular, the one offered by multi-core machines) even in the case of the programming of an unique application. Third, efficiency is an important concern, that was present all along the language design.

The FunLoft language[1] is based on the FairThread model[10] for concurrency aspects, on the PACT language[13] for safety concerns, and on the Synchronous π-calculus model[4] for resource control aspects.

1.1 Overview of FunLoft

A program in FunLoft is a list of definition of variables, types, functions, and module, in a syntax inspirated from ML. In order to be executed, a program must contain the definition of a module named `main` which is the program entry-point. A variable is a name to which a value is associated. Amongst the values are the standard ones (booleans and integers, for example), the values created from type definitions, and the threads, the events, and the schedulers.Type definitions introduce structured data built from union and concatenation of others types. Types can be recursively defined, to define data with variable sizes (lists, for example). Functions are first-order only: they cannot take functions as parameters, nor return functions; moreover it is not possible to define local or anonymous functions. Functions can be recursively defined; however, recursion is checked for termination, in a sense explained below, and thus function calls always terminate.

Modules Modules are templates from which threads, called *instances*, are created. A module can have parameters which define corresponding parameters of its instances. Arguments provided when an instance is created are associated to these parameters. A module can also have local variables, new fresh instances of which are automatically created for each thread created from it. As opposite to functions, modules cannot be recursively defined. The body of a module is basically a sequence of instructions executed by its instances. There are two types of instructions: atomic instructions and non-atomic ones. Atomic instructions are logically run in one single step. Function calls belong to this kind of instructions. Execution of non-atomic instructions may need several steps up to completion. This is typically the case of the instruction which waits for an event to be generated (see below). Execution steps of non-atomic instructions are interleaved, and can thus interfere during execution.

Schedulers A scheduler defines a portion of the memory sharable by the threads linked to it. A special scheduler (the *implicit scheduler*) is automatically launched by each executable program and a thread created from the main module is run in it. The basic task of a scheduler is to control execution of the threads linked to it. The scheduling is cooperative: linked threads have to return the control to the scheduler to which they are linked, in order to let other threads execute. Leaving the control can be either explicit, with the instruction

`cooperate`, or implicit, by waiting for an event which is not present. All linked threads are cyclically considered in turn by the scheduler until all of them have reached a cooperation point (`cooperate`, or waiting instructions). Then, and only then, a new cycle can start. Cycles are called *instants*. A scheduler thus define an automatic synchronization mechanism which forces the threads linked to it to run at the same pace: all the threads must have finished their execution for the current instant, before the next instant can start. Note that the same thread can receive the control from the scheduler several times during the same instant; this is for example the case when the thread waits for an event which is generated by another thread later in the same instant. In this case, the thread receives the control a first time and then blocks, waiting for the event. The control goes to the others threads, and returns back to the first thread after the generation of the event. At creation, each thread is linked to one scheduler (by default, the implicit scheduler). Several schedulers can be defined and simultaneously running in the same program. Schedulers thus define synchronized areas in which threads execute in cooperation. Basically, schedulers run autonomously, in a preemptive way under the supervision of the OS (of course, this has a meaning only in the context of a preemptive OS). During their execution, threads can unlink from the scheduler to which they are currently linked, and become free from any scheduler synchronization. Such free unlinked threads are, like schedulers, run by kernel threads under supervision of the OS. There is a way to define *synchronised schedulers* that share the same instants.

Communication and Synchronisation The simpler way for threads to communicate is of course to use shared variables. For example, a thread can set a boolean variable to indicate that a condition is set, and the other threads can test the variable to know the status of the condition. This basic pattern works well when all threads accessing the variable are linked to the same scheduler. Indeed, in this case atomicity of the accesses to the variable is guaranteed by the cooperativeness of the scheduler. A general way to protect a data from concurrent accesses is thus to associate it with a scheduler to which threads willing to access the data should first link to. Events are synchronizing data basically used by threads to avoid busy-waiting on conditions. An event is always associated to a scheduler (by default, the implicit scheduler) which is in charge of it during all its lifetime. An event is either present or absent during each instant of the scheduler which manages it. It is present if it is generated by the scheduler at the beginning of the instant, or if it is generated by one of the thread executed by the scheduler during the instant; it is absent otherwise. The presence or the absence of an event can change from an instant to another, but all threads always "see" presence or absence of events in the same way, independently on the order in which the threads are scheduled. This is what is meant by saying that events are *broadcast*. Values can be associated to event generations; they are collected during each instant and their collection as a list becomes available at the next instant.

Events are shared by synchronised schedulers, which is possible because syn-

chronised schedulers themselves share instants. All works as if declaring an event in one of the synchronised scheduler automatically declares a corresponding event in the other synchronised schedulers. Moreover, each generation of one of these event is also automatically transmitted to all the associated events.

Memory Model The memory is divided in several parts:

- A private memory for each thread. This memory is initialised when the thread is created, form the parameters and the local variables of the module from which the thread is created. The system checks that no other thread can have access to this private memory.
- A private memory for each scheduler. This memory can only be accessed by the thread linked to the scheduler. The system checks that it is inaccessible from unlinked threads, or from threads linked to another scheduler.

Thus, there exist no global variable shared by distinct schedulers, and a variable that is shared by distinct threads belongs to a unique scheduler and can only be accessed by the threads linked to the scheduler. This is the way data-races are absent from FunLoft.

1.2 Overview of Static Analysis

Several properties are statically checked in FunLoft:

- Functions always terminate (despite the fact that they can be recursively defined).
- Linked threads always cooperate(despite that they are allowed to never terminate).
- The memory model described above is actually respected.
- The number of simultaneously living threads always stays under control (despite the fact that threads can be dynamically created).
- No reference is allowed to increase in an uncontrolled way.

One thus gets a concurrent language in which programs are proved to be free from data-races and memory leaks.

For linked threads, the model guarantees the logical atomicity of the sequence of two instructions. If a thread contains the sequence A;B then no other thread can logically insert itself between A and B, *even if this is physically possible*.

To end the overview of FunLoft, we introduce its syntax by means of examples of programs that are statically accepted or rejected.

Instantaneous Loops Here is the definition of a correct module `m` which at each instant prints a message given as argument and cooperates:

```
let module m (msg) =
    while true do begin print_string (msg); cooperate end
```

On the contrary, the following module is incorrect as it does not cooperate:

```
let module m (msg) =
    while true do print_string (msg)
```

Linking a thread instance of it to a scheduler would prevent from execution all other threads linked to the scheduler. This module is said to have an *instantaneous loop*. This point will be discussed in section 4.

Data Races The following program is correct: atomicity of the assignment is guaranteed because the two threads created are executed cooperatively. A data-race (on `r`) is thus impossible:

```
let r = ref 0
let module m (n) = r:=n
let module main () =
   begin
      thread m (1);
      thread m (2);
   end
```

On the contrary, the following program is incorrect because the reference `r` is accessed by a thread linked to the implicit scheduler, and also by an unlinked thread:

```
let r = ref 0
let module m (n) = r:=n
let module main () =
   begin
      unlink thread m (1);
      thread m (2);
   end
```

Indeed, the two threads are mapped on two native treads which are under the control of the OS. Thus, one has no guarantee of atomic access to `r`, which introduces a possibility of data-race.

A loss of atomicity also appears in the following program, where `r` is accessed by two threads linked to two distinct schedulers (`s` and the implicit scheduler):

```
let s = scheduler
let r = ref 0
let module m (n) = r:=n
let module main () =
   begin
      link s do thread m (1);
      thread m (2);
   end
```

Transmission of private reference In the following example, the reference `r` created in `m` is necessarilly private, because it is read while unlinked. Thus, it should not be accessible from the public reference `g`. But this access is made possible by the assignement `g:=r`, and the program is thus incorrect:

```
let g = ref ref 0
let module m () =
  let r = ref 0 in
     begin
        unlink !r;
        g:=r;
     end
```

Note that, if this program were accepted, it would be very easy to define a context that would produce a data-race.

Number of threads Situations were the number of threads to be executed (the living threads) is always increasing should be forbidden. This is for example the case with the following program:

```
while true do
  begin
    thread m ();
    cooperate;
  end
```

Indeed, suppose `m` never terminates, then the number of living threads would be incremented at each instant, which would eventually leads to a memory overflow. The program should thus be rejected.

Note however that the program becomes acceptable if synchronous thread creation (`run` primitive) replaces asynchronous creation:

```
while true do
  begin
    run m ();
    cooperate;
  end
```

Assuming that the number of threads which can be created by `m` is bounded by n, one is sure that the loop will never create more than n simultaneously living threads.

Size of data To control the size of references basically means to forbid the use of references as accumulators. Typical example of accumulator is:

```
type list = Nil | Cons of int * list
let module m () =
  let r = ref Nil in
     while true do
```

```
        begin
          r := Cons (0,!r);
          cooperate;
        end
```

The content of the reference `r` is a list whose length becomes more and more larger. Execution of this program will eventually lead to a memory overflow.

Note that the accumulation phenomenon may involve several references, as in:

```
let r1 = ref Nil
let r2 = ref Nil
let module m (r1,r2) =
   while true do
      begin
         r1 := Cons (0,!r2);
         cooperate;
      end
let module main () =
   begin
      thread m (r1,r2);
      thread m (r2,r1);
   end
```

1.3 Structure of the Report

One first considers in section 2 a basic formalism, without data, covering thread creation, execution, joining, and migration. Scheduler synchronisation is also treated in this basic fragment. One gives an operational semantics for the basic formalism. Data and references are introduced ins section 3. Data and functions to manipulate them are considered at an abstract level, without polymorphism. The issue of instantaneous loops is considered in section 4. A type system to eliminate data-races is given in section 5. Section 6 describes a variant of the operational semantics in which violations of the memory model appear explicitely. A well-typed program never leads to such violations. Inference of reference status is considered in section 7. A program for which one can infer a coherent information is proved well typed, thus free from data-races. Resource control is considered in section 8. This section contains two type systems: one for memory stratification, and the other to control the number of simultaneously living threads. Section 9 overview the issue of termination detection of functions. Events are introduced in section 10 which describes the changes induced in the operational semantics and the static analysis. Several features not yet formalised are considered in section 11. Finally, related work is described in section 12.

2 Basic Fragment

In a first step, one considers a basic fragment which only captures thread creation, schedulers synchronisation, and migration. The `join` primitive of FunLoft is considered under the restricted form of a `run` statement which creates a thread and waits for the termination of all the threads transitively created. The unlinking of a thread is seen as the migration to a new anonymous scheduler dedicated to its execution.

One considers an enumerable set of scheduler names, augmented with the special symbol _ for anonymous schedulers: $\mathcal{S}_{name} ::= \{\_\} \cup \{l_1, l_2, \cdots\}$.

The syntax of the basic fragment is:

Syntax $P ::= \mathbf{0} \mid A \mid \mu A.P \mid P; P \mid P@n \mid \texttt{thread}\ P \mid \texttt{run}\ P \mid \texttt{cooperate}$
More precisely:

- $\mathbf{0}$ is the terminated program.
- $\mu A.P$ is the recursive definition of P, with A as recursion variable.
- $P_1; P_2$ is the sequence of P_1, followed by P_2. The program P_2 starts when P_1 terminates.
- $P@l$ is the migration to the scheduler l for running P. After P termination, the inverse migration takes place. $P@\_$ is the execution of P as unlinked.
- `thread` P creates a new thread running P and terminates instantly.
- `run`P creates a new thread running P and terminates when all the threads transitively created by P, or by threads created by it, are terminated.
- `cooperate` terminates execution for the current instant; execution restarts in sequence from the `cooperate` instruction at the next instant.

Threads A thread is a couple (t, P) where t is a thread name. One notes the thread (t, P) by P_t.

Scheduler A scheduler is a triple: $S ::= < L, L' >_n$ where L is a list of active threads, L' is a list of threads to be considered at the next instant, and n is a scheduler name. Schedulers with name _ are called *anonymous* schedulers. Actually, there are two kinds of elements in L':

- standard threads of the form P_t;
- guarded threads of the form $t \uparrow P_{t'}$ meaning that termination of thread t and of all its descendants (threads transitively created by t) is required to continue execution of $P_{t'}$.

If X is a list of threads, one notes $x \cdot X$ the list with head x and tail X.

One supposes that there exists an equivalence relation over schedulers: the relation of synchronisation. If S is a scheduler, the equivalence class of S (the maximal set of schedulers that are transitively synchronised with S) is noted $\widehat{S}$. From the language definition, anonymous schedulers cannot be synchronised with others schedulers (unlinked threads are autonomous).

Evaluation Context An evaluation context has form:
$E ::= [\,] \mid E@n \mid E; P$
The predicate $E \looparrowright n$ is true if n appears in E and is the left-most (most intern) scheduler name in it (for example, $[\,]@n_1@n_2 \looparrowright n_1$).

Global Context A global context has form: $C ::= [\,] \mid (C|C) \mid S$

Creation Tree One notes $\mathcal{T}$ the tree that records the created threads and their state (terminated or not); in this tree, an arc from t to t' means that t' has been created by t. Using $\mathcal{T}$, one can determine if a thread is terminated or not, and if all threads created from it are terminated or not. This information is useful for the treatment of the `run` primitive. One notes $over_{\mathcal{T}}(t)$ if t is a terminated node of the tree $\mathcal{T}$ and if all its decendants are also transitively terminated. Note that in the sequel, $\mathcal{T}$ is always growing: nodes are never removed from it. When t is a node of $\mathcal{T}$ and t' is a new name not appearing in $\mathcal{T}$, one notes $\mathcal{T} \cup \{t \rightarrow t'\}$ the tree in which the node t' is added as a new son of t.

End of Instant Relation The $\downarrow_{eoi}^{\mathcal{T}}$ relation is defined by:

$$< \emptyset, L' >_n \downarrow_{eoi}^{\mathcal{T}} < X, Y >_n$$

where X is the list of threads obtained from elements of L' which are standard threads or are threads guarded by terminated nodes, and Y is the sublist of L' of elements guarded by non-terminated nodes:

- X is the list of threads extracted from elements of L' verifying the predicate:
  $Q(e) \equiv\ e = P_t \vee (e = t \uparrow P_{t'}\ \wedge\ over_{\mathcal{T}}(t))$
- Y is the sub-list of elements of L' verifying the predicate:
  $R(e) \equiv\ e = t \uparrow P_{t'}\ \wedge\ \neg\ over_{\mathcal{T}}(t)$

Global Rules A thread executes in the scheduler to which it is linked:

$$\frac{P \to P' \quad E \looparrowright n}{C[< E[P]_t \cdot L, L' >_n], \mathcal{T} \to C[< E[P']_t \cdot L, L' >_n], \mathcal{T}}$$

Synchronised schedulers all synchronise, in one unique step, at the end of each instant:

$$\frac{S_i \downarrow^{\mathcal{T}}_{eoi} S'_i \quad S_i \in \widehat{S_1} = \{S_1, \ldots, S_k\}}{C[S_1] \ldots [S_k], \mathcal{T} \to C[S'_1] \ldots [S'_k], \mathcal{T}}$$

Migration to a specific scheduler extracts the thread from the source scheduler and places it into the target scheduler:

$$\frac{E \looparrowright l \quad n \neq l}{C[< E[P]_t \cdot L, L' >_n][< L_1, L_2 >_l], \mathcal{T} \to C[< L, L' >_n][< L_1, L_2 \cdot E[P]_t >_l], \mathcal{T}}$$

The unlinking of a thread is seen as the migration to a new anonymous scheduler dedicated to its execution:

$$\frac{E \looparrowright \_}{C[< E[P]_t \cdot L, L' >_l], \mathcal{T} \to C[< L, L' >_l][< E[P]_t, \emptyset >_\_], \mathcal{T}} \quad (1)$$

Local Rules In the recursive definition of P, the recursion variable is substituted by its definition:

$$\mu A.P \to P[\mu A.P/A]$$

When the left part of a sequence is terminated, then the control goes to the right part:

$$\mathbf{0}; P \to P$$

Thread Rules Terminated threads are eliminated from the scheduler to which they are linked:

$$C[< \mathbf{0}_t \cdot L, L' >_n], \mathcal{T} \to C[< L, L' >_n], \mathcal{T}'$$

where $\mathcal{T}'$ is $\mathcal{T}$ in which the state of t is set to terminated.
An asynchronous thread creation puts the new created thread in the list of thread to be considered at next instant:

$$\frac{E \looparrowright l}{C[< E[\texttt{thread}\ P]_t \cdot L, L' >_l], \mathcal{T} \to C[< E[\mathbf{0}]_t \cdot L, L' \cdot (P@l)_{t'} >_l], \mathcal{T}'} \quad (2)$$

where t' is a new node and $\mathcal{T}' = \mathcal{T} \cup \{t \to t'\}$.
Synchronous thread creation is similar to asynchronous thread creation, except that the new created thread is guarded by termination of the initial thread:

$$\frac{E \not\mapsto l}{C[< E[\texttt{run}\ P]_t \cdot L, L' >_l], \mathcal{T} \to C[< L, L' \cdot (P@l)_{t'} \cdot t' \uparrow E[\mathbf{0}]_t >_l], \mathcal{T}'} \tag{3}$$

where t' is a new node and $\mathcal{T}' = \mathcal{T} \cup \{t \to t'\}$.

Cooperation Rule Cooperation terminates execution for the current instant and transfers the continuation in the list of threads to be executed at next instant:

$$\frac{E \not\mapsto l}{C[< E[\texttt{cooperate}]_t \cdot L, L' >_l], \mathcal{T} \to C[< L, L' \cdot E[\mathbf{0}]_t >_l], \mathcal{T}} \tag{4}$$

Errors There are several situation where no rule apply to a given program. These situations are considered as errors:

- Migration to a scheduler which does not exist.
- Unlinking while already unlinked (rule 1; for example in: $(P@\_)@\_$).
- Thread creation while unlinked (rules 2 and 3).
- Cooperation while unlinked (rule 4).

In the rest of the text, programs are supposed to be free from these errors (they are rejected either directly by the syntax, or by an elementary typing not considered here).

3 References

In a second step, one extends the previous basic fragment with data and references to hold them.

Data are either basic data (e.g. booleans) or structured data belonging to *inductive* data types. An inductive data type is defined by a list of constructors. For example, the type *intlist* of lists of integers is defined by:

$type\ intlist = Nil \mid Cons\ of\ int * intlist$

which introduces two constructors Nil and $Cons$. A specific matching instruction is used to decompose data of inductive types, according to the way they have been built.

Functions are defined for data processing. Functions are first-order (no function as parameter; no function returned). One makes a strong assumption: function calls always terminate instantly, independently of the values of the actual

parameters. An example of function is the one that returns the length of a list of integers.

One considers *modules* which are defined by definitions of the form $A(\overline{x}) = P$, where the notation $\overline{x}$ is a short-hand for the (possibly empty) list $x_1, \cdots, x_n$. Variables $\overline{x}$ are the module parameters which can appear in the module body P. Recursivity is forbidden in module definitions but a loop instruction is provided to define cyclic behaviours.

Syntax The syntax of programs becomes:

$P ::= \mathbf{0} \mid P; P \mid P@n \mid \texttt{thread}\ A(\overline{e}) \mid \texttt{run}\ A(\overline{e}) \mid \texttt{cooperate} \mid \texttt{let}\ x = e\ \texttt{in}\ P$
$\mid e := e \mid [e \rhd p]P, P \mid \texttt{while}\ e\ \texttt{do}\ P$

Differences with previous syntax are:

- $\texttt{thread}\ A(\overline{e})$, where A is defined by $A(\overline{x}) = P$, creates a new thread running P with parameters $\overline{x}$ given values $\overline{e}$, and terminates instantly. $\texttt{run}\ A(\overline{e})$ is similar, but waits for termination of P and of all the threads transitively created by it.
- The definition $\texttt{let}\ x = e\ \texttt{in}\ P$ associates the value of e to the variable x in P.
- The assignment $e_1 := e_2$ assigns the value of e_2 to the reference denoted by e_1.
- Instruction $[e \rhd p]P_1, P_2$ is an attempt to decompose the value denoted by e following the pattern p. If the decomposition is possible (one supposes, then, that it is unique), then P_1 is run in the environment resulting from the decomposition. Otherwise, P_2 is run.
- The loop $\texttt{while}\ e\ \texttt{do}\ P$ executes P cyclically while the expression e is true.

A reference creation extends the heap with a new fresh location:

$$\frac{e, \mathcal{H} \Downarrow v, \mathcal{H}' \quad r \notin Dom(\mathcal{H}')}{\texttt{ref}\ e, \mathcal{H} \Downarrow r, \mathcal{H}' \oplus [r \backslash v]}$$

Evaluation of a location returns it:

$$r, \mathcal{H} \Downarrow r, \mathcal{H}$$

Dereferencing a reference returns its content:

$$\frac{e, \mathcal{H} \Downarrow r, \mathcal{H}'}{!e, \mathcal{H} \Downarrow \mathcal{H}'(r), \mathcal{H}'}$$

Global Rules There are no real changes for the global rules, except the introduction of the heap.

$$\frac{P, \mathcal{H} \rightarrow P', \mathcal{H}' \quad E \not\rightarrow n}{C[< E[P]_t \cdot L, L' >_n], \mathcal{T}, \mathcal{H} \rightarrow C[< E[P']_t \cdot L, L' >_n], \mathcal{T}, \mathcal{H}'} \tag{5}$$

$$\frac{S_i \downarrow_{eoi}^{\mathcal{T}} S_i' \quad S_i \in \widehat{S_1} = \{S_1, \ldots, S_k\}}{C[S_1] \ldots [S_k], \mathcal{T}, \mathcal{H} \rightarrow C[S_1'] \ldots [S_k'], \mathcal{T}, \mathcal{H}}$$

$$\frac{E \not\rightarrow l \quad n \neq l}{C[< E[P]_t \cdot L, L' >_n][< L_1, L_2 >_l], \mathcal{T}, \mathcal{H} \rightarrow C[< L, L' >_n][< L_1, L_2 \cdot E[P]_t >_l], \mathcal{T}, \mathcal{H}}$$

$$\frac{E \not\rightarrow \_}{C[< E[P]_t \cdot L, L' >_l], \mathcal{T}, \mathcal{H} \rightarrow C[< L, L' >_l][< E[P]_t, \emptyset >_\_], \mathcal{T}, \mathcal{H}}$$

Local Rules The heap is introduced in the rule for the sequence instruction:

$$\mathbf{0}; P, \mathcal{H} \rightarrow P, \mathcal{H}$$

The binding instruction is standard:

$$\frac{e, \mathcal{H} \Downarrow v, \mathcal{H}'}{\texttt{let}\ x = e\ \texttt{in}\ P, \mathcal{H} \rightarrow P[x \backslash v], \mathcal{H}'}$$

An assignement changes the heap and rewrites in $\mathbf{0}$:

$$\frac{(e_1, e_2), \mathcal{H} \Downarrow (r, v), \mathcal{H}'}{e_1 := e_2, \mathcal{H} \rightarrow \mathbf{0}, \mathcal{H}'[r \backslash v]}$$

There are two cases for the matching instruction, depending on the possibility to match the value of the expression with the pattern (the matching is possible if there exists a substitution σ which applied to the pattern gives the value):

$$\frac{e, \mathcal{H} \Downarrow v, \mathcal{H}' \quad \exists \sigma . \sigma(p) = v}{[e \rhd p] P_1, P_2, \mathcal{H} \rightarrow \sigma(P_1), \mathcal{H}'}$$

$$\frac{e, \mathcal{H} \Downarrow v, \mathcal{H}' \quad \nexists \sigma . \sigma(p) = v}{[e \rhd p] P_1, P_2, \mathcal{H} \rightarrow P_2, \mathcal{H}'}$$

A loop first evaluates the controlling expression and terminates when the returned value is false:

$$\frac{e, \mathcal{H} \Downarrow \texttt{false}, \mathcal{H}'}{\texttt{while}\ e\ \texttt{do}\ P, \mathcal{H} \rightarrow \mathbf{0}, \mathcal{H}'}$$

Otherwise, the loop rewrites in a sequence:

$$\frac{e, \mathcal{H} \Downarrow \mathit{true}, \mathcal{H}' \quad P, \mathcal{H}' \rightarrow P', \mathcal{H}''}{\texttt{while}\ e\ \texttt{do}\ P, \mathcal{H} \rightarrow P'; \texttt{while}\ e\ \texttt{do}\ P, \mathcal{H}''} \tag{6}$$

Thread Rules Thread rules are very similar to the ones of the basic fragment:

$$C[< \mathbf{0}_t \cdot L, L' >_n], \mathcal{T}, \mathcal{H} \rightarrow C[< L, L' >_n], \mathcal{T}', \mathcal{H}$$

$$\frac{E \not\hookrightarrow l \quad \overline{e}, \mathcal{H} \Downarrow \overline{v}, \mathcal{H}'}{C[< E[\texttt{thread}\ A(\overline{e})]_t \cdot L, L' >_l], \mathcal{T}, \mathcal{H} \rightarrow C[< E[\mathbf{0}]_t \cdot L, L' \cdot (P[\overline{x} \backslash \overline{v}]@l)_{t'} >_l], \mathcal{T}', \mathcal{H}'} \tag{7}$$

$$\frac{E \not\hookrightarrow l \quad \overline{e}, \mathcal{H} \Downarrow \overline{v}, \mathcal{H}'}{C[< E[\texttt{run}\ A(\overline{e})]_t \cdot L, L' >_l], \mathcal{T}, \mathcal{H} \rightarrow C[< L, L' \cdot (P[\overline{x} \backslash \overline{v}]@l)_{t'} \cdot t' \uparrow E[\mathbf{0}]_t >_l], \mathcal{T}', \mathcal{H}'} \tag{8}$$

Cooperation Rule The cooperation rule is the same, except introduction of the heap:

$$\frac{E \not\hookrightarrow l}{C[< E[\texttt{cooperate}]_t \cdot L, L' >_l], \mathcal{T}, \mathcal{H} \rightarrow C[< L, L' \cdot E[\mathbf{0}]_t >_l], \mathcal{T}, \mathcal{H}}$$

Errors Two new possibilities of errors appear:

1. Incomplete match instructions (there are uncovered cases), or match instructions with not disjoint cases.
2. Loop instructions in which the body does not converge; then, no rewriting exists. This kind of error is called "instantaneous loop" in synchronous languages, such as Esterel[8]. Elimination of instantaneous loops is considered in section 4.

In the following, programs are supposed to be free from the first kind of errors, according to a syntax analysis which, for simplicity, is not described here.

4 Instantaneous Loops

One types programs by $P\|\!\vdash b$, where b is a boolean indicating the possible instantaneous termination of P. One notes $P\|\!\vdash \mathbf{true}$ if P can terminate instantly, and $P\|\!\vdash \mathbf{false}$ when P never terminates instantly.

Always Instantly Terminating

$$\mathbf{0}\|\!\vdash \mathbf{true}$$

$$e_1 := e_2\|\!\vdash \mathbf{true}$$

$$\texttt{thread}\ A(\overline{e})\|\!\vdash \mathbf{true}$$

Never Instantly Terminating

$$\texttt{cooperate}\|\!\vdash \mathbf{false}$$

$$\texttt{run}\ A(\overline{e})\|\!\vdash \mathbf{false}$$

Sequence

$$\frac{P_1\|\!\vdash b_1 \quad P_2\|\!\vdash b_2}{P_1; P_2\|\!\vdash b_1 \wedge b_2}$$

Match

$$\frac{P_1\|\!\vdash b_1 \quad P_2\|\!\vdash b_2}{[e \rhd p]P_1, P_2\|\!\vdash b_1 \vee b_2}$$

Variable and Migration

$$\frac{P\|\!\vdash b}{\texttt{let}\ x = e\ \texttt{in}\ P\|\!\vdash b}$$

$$\frac{P\|\!\vdash b}{P@n\|\!\vdash b}$$

Loop The body of a loop should never terminate instantly. The whole loop however can terminate instantly (the controling expression could be false):

$$\frac{P\|\vdash \mathbf{false}}{\texttt{while}\ e\ \texttt{do}\ P\|\vdash \mathbf{true}}$$

Proposition 1 *Let P a program such that $P\|\vdash b$. Then, for all $\mathcal{H}$, there exist P' and $\mathcal{H}'$ such that $P, \mathcal{H} \rightarrow P', \mathcal{H}'$ and $b = \texttt{false}$ implies $P' \neq \mathbf{0}$.*

As a consequence, if $P\|\vdash b$ then no instantaneous loop can appear during execution of P (application of the recursive rule 6 cannot cycle). Thus $P\|\vdash b$ means that P is free from instantaneous loops.

5 Type System for Atomicity

One now considers typing with the objective to detect data-races.

Data types are either the unit type (whose only value is noted "()"), type names (denoting basic types or inductively defined types) or reference types of the form $\tau\ \texttt{ref}_n$ with $n \in \mathcal{S}_{name}$. Type $\tau\ \texttt{ref}_\_$ is the type of *private* references; a private reference should only be accessed by the thread which has created it. Type $\tau\ \texttt{ref}_l$ is the type of *public* references which are shared by all the threads linked to the scheduler l. Types are defined by:

$\tau ::= () \mid t \mid \tau\ \texttt{ref}_n$

A typing environment Γ is a possibly empty list of elements of the form $x : \tau$, where x is a type variable:

$\Gamma ::= x_1 : \tau_1, \cdots, x_n : \tau_n$

An *effect* F is a set of scheduler names; $l \in F$ denotes an access to a reference belonging to scheduler l. The auxiliary function *actual* is defined by $actual(\_) = \emptyset$, and $actual(l) = \{l\}$.

Type judgments have the form $\Gamma \vdash P : F$, for programs, and $\Gamma \vdash e : \tau, F$, for expressions.

One says that a type τ is public, noted $public(\tau)$, if τ is not of the form $\tau'\ \texttt{ref}\_$.

The type information associated with a function f is noted $f : \overline{\tau} \rightarrow \tau'/F$ to indicate that the function has parameters of types $\overline{\tau}$, returns a value of type τ', and produces effect F. For example, the type $int\ \texttt{ref}_l \rightarrow int/\{l\}$ is a possible type for a function defined by $\texttt{let}\ f(x) =!x$. Note that polymorphism (parametric types) is not considered here.

The type information associated with a module A is noted $A : \overline{\tau} \rightarrow ()/F$ to indicate that the module has parameters of types $\overline{\tau}$ and produces effect F (modules do not return values or, alternatively, they return "()").

5.1 Type System

Programs The termination instruction has no effect:

$$\Gamma \vdash \mathbf{0} : \emptyset$$

The effects of the control expression and of the body are collected in a loop:

$$\frac{\Gamma \vdash P : F_1 \quad \Gamma \vdash e : bool, F_2}{\Gamma \vdash \texttt{while}\, e\, \texttt{do}\, P : F_1 \cup F_2}$$

The effects of the two components of a sequence are collected:

$$\frac{\Gamma \vdash P_i : F_i}{\Gamma \vdash P_1; P_2 : \cup F_i}$$

No effect is allowed when unlinked. The only effect allowed when linked to an explicit scheduler is the access to references belonging to the scheduler:

$$\frac{\Gamma \vdash P : F \quad F \subseteq actual(n)}{\Gamma \vdash P@n : \emptyset}$$

In a thread creation, parameters should be public. The effect of the creation is the collection of the effects of the actual parameters:

$$\frac{A : \overline{\tau} \to ()/F \quad \Gamma \vdash e_i : \tau_i, F_i \quad public(\tau_i)}{\Gamma \vdash \texttt{thread}\ A(\overline{e}) : \cup F_i}$$

Synchronous and asynchronous thread creations are treated exactly in the same way:

$$\frac{A : \overline{\tau} \to ()/F \quad \Gamma \vdash e_i : \tau_i, F_i \quad public(\tau_i)}{\Gamma \vdash \texttt{run}\ A(\overline{e}) : \cup F_i}$$

The cooperation instruction has no effect:

$$\Gamma \vdash \texttt{cooperate} : \emptyset$$

The body of a variable definition is analysed in a context in which the variable is defined; the effect combines the effect produced by the initial value of the variable with the effect of the body:

$$\frac{\Gamma \vdash e : \tau, F_1 \quad \Gamma \cup x : \tau \vdash P : F_2}{\Gamma \vdash \texttt{let}\, x = e\, \texttt{in}\, P : F_1 \cup F_2}$$

Assignment to a reference adds to the effect the scheduler associated with the reference:

$$\frac{\Gamma \vdash e_1 : \tau\ \texttt{ref}_n, F_1 \quad \Gamma \vdash e_2 : \tau, F_2}{\Gamma \vdash e_1 := e_2 : F_1 \cup F_2 \cup actual(n)}$$

Effects of the two alternatives of a matching statement are collected in the final effect:

$$\frac{\Gamma \vdash e : \tau, F_0 \quad \Gamma \cup p : \tau \vdash P_1 : F_1 \quad \Gamma \vdash P_2 : F_2}{\Gamma \vdash [e \rhd p]P_1, P_2 : F_0 \cup F_1 \cup F_2}$$

Expressions Variables can have any type:

$$\Gamma \cup x : \tau \vdash x : \tau, \emptyset$$

Effects of parameters and effect of the function are collected in a function call:

$$\frac{f : \overline{\tau} \to \tau'/F \quad \Gamma \vdash e_i : \tau_i, F_i}{\Gamma \vdash f(\overline{e}) : \tau', F \cup F_i}$$

Constructors have no effect except those of their arguments:

$$\frac{C : \overline{\tau} \to t/\emptyset \quad \Gamma \vdash e_i : \tau_i, F_i}{\Gamma \vdash C(\overline{e}) : t, \cup F_i}$$

For a public reference, the type of its initial value should be public:

$$\frac{\Gamma \vdash e : \tau, F \quad n \neq \_ \Rightarrow \mathit{public}(\tau)}{\Gamma \vdash \mathtt{ref}_n e : \tau\ \mathtt{ref}_n, F}$$

Dereferencing a reference adds to the effect the scheduler associated with the reference:

$$\frac{\Gamma \vdash e : \tau\ \mathtt{ref}_n, F}{\Gamma \vdash\ !e : \tau, F \cup \mathit{actual}(n)}$$

5.2 Private References

In a well-typed program, a thread cannot access the private references belonging to another thread. To prove this, one introduces the notion of a coherent heap and shows that coherency is preserved by reductions. Communicating a private reference to an other thread needs, at some state, the heap to be incoherent, which gives the result.

A heap $\mathcal{H}$ is *coherent* if no public reference points on a private one: $\mathcal{H}(r_l) = r'_n$ implies $n \neq \_$. First, one proves the following proposition:

Proposition 2 *Heap coherency is preserved through evaluation.*

Let e be a closed expression and suppose $\mathcal{H}$ is coherent. If $\vdash e : \tau, F$ and $e, \mathcal{H} \Downarrow v, \mathcal{H}'$, then $\mathcal{H}'$ is coherent.

Expression e cannot be a variable or a reference as it would not be closed.

Consider reference creation. Suppose $\vdash \mathtt{ref}_n e : \tau\ \mathtt{ref}_n, F$ and

$$\frac{e, \mathcal{H} \Downarrow v, \mathcal{H}' \quad r \notin \mathit{Dom}(\mathcal{H}')}{\mathtt{ref}_n e, \mathcal{H} \Downarrow r, \mathcal{H}' \oplus [r \backslash v]}$$

One has $\vdash e : \tau, F$. By induction, $\mathcal{H}'$ is coherent. Let $\mathcal{H}'' = \mathcal{H}' \oplus [r \backslash v]$. Suppose $\mathcal{H}''$ is not coherent. This means that $n \neq \_$ and $\mathcal{H}''(r_n) = v = r'_{\_}$. But, then $\tau = \tau'$ `ref`$_{\_}$ which would contradict the fact that $public(\tau)$. Thus, $\mathcal{H}''$ is coherent.

The other cases (dereferencing, function call, and constructor call) are immediate.

Heap coherency is also preserved through reduction: if $\mathcal{H}$ is coherent, $P, \mathcal{H} \rightarrow P', \mathcal{H}'$, and $\emptyset \vdash P : F$, then $\mathcal{H}'$ is coherent (the proof is concerned only with assignment; the result comes from the preservation of the type of the left part).

Finally, heap consistency is preserved by system reductions (immediate, from previous cases).

□

As consequence of heap coherency preservation, a private reference of a thread cannot be shared. Indeed, for a thread to give access to a private reference to another thread means either to pass it as parameter (which is impossible because parameters have to be public), or to copy the private reference into a public reference, which leads to an incoherent heap.

6 Controlled Execution Semantics

One defines a variant of the execution semantics of section 3 in which accesses to references are controlled, in the sense that "wrong" accesses, leading to possibilities of data-races, produce errors (no rule apply).

Expressions An *owner* is associated to each reference value. There are two cases: either the owner is a scheduler name which means that the reference is public, that is, sharable by the threads linked to the scheduler; either the owner is a thread name which means that the reference is private to this thread, and thus can only be accessed by it.
Expressions are defined by:

$e ::= x \mid r_o \mid f(\overline{e}) \mid C(\overline{e}) \mid \texttt{ref}_n\ e \mid !e$

In reference r_o, o is the owner of the reference.

Executing Thread and Current Scheduler Rules 5, 7, and 8 are those in which expressions are evaluated and programs are reduced in a given context. They have the shape:

$C[< E[\cdots]_t, \cdots >_n], \mathcal{T}, \mathcal{H} \rightarrow C[\cdots], \mathcal{T}', \mathcal{H}'$

Actually, t is the executing thread, and n is the current scheduler (possibly $\_$, when t is unlinked). Rules are, if needed, prefixed by $\vdash^t_l$ when there are used by these 3 rules.

6.1 Operational Semantics

Evaluation of Expressions Creation of a public reference in a scheduler l extends the heap with a new fresh location created in l. Creation is impossible

if the current scheduler is different from l:

$$\vdash_l^t \frac{e, \mathcal{H} \Downarrow v, \mathcal{H}' \quad r_l \notin Dom(\mathcal{H}')}{\texttt{ref}_l\, e, \mathcal{H} \Downarrow r_l, \mathcal{H}' \oplus [r_l \backslash v]}$$

Creation of a private reference (with _) creates a reference the owner of which is the executing thread. Creation is possible independently of the fact that the thread is linked or unlinked:

$$\vdash_n^t \frac{e, \mathcal{H} \Downarrow v, \mathcal{H}' \quad r_t \notin Dom(\mathcal{H}')}{\texttt{ref}_{\_}\, e, \mathcal{H} \Downarrow r_t, \mathcal{H}' \oplus [r_t \backslash v]}$$

Evaluation of a direct reference returns it:

$$r_n, \mathcal{H} \Downarrow r_n, \mathcal{H}$$

Dereferencing a reference is only possible if it is performed in the correct context: if the reference is private, the thread must be the owner of the reference:

$$\vdash_n^t \frac{e, \mathcal{H} \Downarrow r_t, \mathcal{H}'}{!e, \mathcal{H} \Downarrow \mathcal{H}'(r_t), \mathcal{H}'}$$

Deferencing a private reference is impossible by a thread not owning it. Dereferencing does not depend of the state linked or unlinked of the thread. If the reference is public, then it should belong to the current scheduler:

$$\vdash_l^t \frac{e, \mathcal{H} \Downarrow r_l, \mathcal{H}'}{!e, \mathcal{H} \Downarrow \mathcal{H}'(r_l), \mathcal{H}'}$$

Local Rules The rules for assignment are very similar to the rules for deferenciation:

$$\vdash_l^t \frac{(e_1, e_2), \mathcal{H} \Downarrow (r_l, v), \mathcal{H}'}{e_1 := e_2, \mathcal{H} \rightarrow \mathbf{0}, \mathcal{H}'[r_l \backslash v]}$$

$$\vdash_n^t \frac{(e_1, e_2), \mathcal{H} \Downarrow (r_t, v), \mathcal{H}'}{e_1 := e_2, \mathcal{H} \rightarrow \mathbf{0}, \mathcal{H}'[r_t \backslash v]}$$

Errors Actually, an execution error is encountered (no rule apply) in two cases:

- a private reference is accessed (dereferenced or assigned) by a thread distinct from the owner of the reference;
- a public reference is accessed by a thread unlinked or not linked to the scheduler to which the reference belongs.

The correction property can now be stated:

Proposition 3 *If a program P is well-typed then there is no execution error for P in the controlled semantics.*

6.2 Absence of Data-races

In a well-typed program, data-races cannot occur. First, a thread linked to a scheduler cannot access a reference belonging to another scheduler. Second, an unlinked thread can only access its private memory. These properties basically result from the typing rule:

$$\frac{\Gamma \vdash P : F \quad F \subseteq actual(n)}{\Gamma \vdash P@n : \emptyset}$$

If $n = \_$, then effect should be empty, thus only private references are accessible. By previous section, these private reference are those of the executing thread. If $n = l$, then the only accessible references are the private ones or the public references belonging to scheduler l. As previously, private references accessed belong to the executing thread. Thus, when a thread is unlinked, it can only have access to its private references, and when a thread is linked to a scheduler it can only have access to the its private reference and to the public references of the scheduler. In all cases, no data-race can occur.

7 Inference System

In this section, we present an inference system to infer the status (private or public, that is belonging to a scheduler) of references.

One defines scheduler variables $\alpha, \beta, \cdots$ and labels each reference creation with a scheduler variable; thus, reference creations get the form: $\texttt{ref}_\alpha e$. All the introduced labels are supposed to be distinct.

One assumes that functions have been correctly typed in a first phase. Types have the shape $f : \forall \overline{\alpha}.\overline{\tau} \to \tau'/F$ where $\overline{\alpha}$ are the scheduler variables appearing in $\overline{\tau}$. For example the type $\forall \alpha.int\ \texttt{ref}_\alpha \to int/\{\alpha\}$ is a possible type for $\texttt{let}\ f(x) = !x$. Note that, here, we consider polymorphism only at the level of scheduler variables, not at the data-type level. For example, the previous function f could as well be given type $\forall \alpha.bool\ \texttt{ref}_\alpha \to bool/\{\alpha\}$.

Inference for programs has the form $\Gamma \models P : F, C$ with C a set of *constraints* of the form $(x = y)$, $(x \neq y)$, or $(x \leq y)$, where x and y are scheduler variables or scheduler names (the set $\mathcal{S}_{name}$). Inference for expressions has the form $\Gamma \models e : \tau, F, C$, where τ is the type of expression e.

The inference process produces a set C of constraints that must be consistent: there should exist a function σ extending the identity on $\mathcal{S}_{name}$ and associating with each scheduler variable an element of $\mathcal{S}_{name}$ such that:

- $(x = y) \in C \Rightarrow \sigma(x) = \sigma(y)$
- $(x \neq y) \in C \Rightarrow \sigma(x) \neq \sigma(y)$
- $(x \leq y) \in C \Rightarrow \sigma(x) = \_ \vee \sigma(x) = \sigma(y)$

Such a function σ is said to be a *unifier* of C.
One defines two functions $NotPriv$ and $Propagate$ by:

- $NotPriv(\tau\ \texttt{ref}_\alpha) = \{(\alpha \neq \_)\}$, and $NotPriv(\tau) = \emptyset$ otherwise.
- $Propagate(\alpha, \tau\ \texttt{ref}_\beta) = \{(\alpha \leq \beta)\}$ and $Propagate(\alpha, \tau) = \emptyset$ otherwise.

We now give the inference system.

Programs

$$\Gamma \models \mathbf{0} : \emptyset, \emptyset$$

$$\frac{\Gamma \models P_i : F_i, C_i}{\Gamma \models P_1; P_2 : \cup F_i, \cup C_i}$$

$$\frac{\Gamma \models P : F, C}{\Gamma \models P@n : \emptyset, C \cup \{(\alpha \leq n) \,/\, \alpha \in F\}}$$

$$\frac{A : \overline{\tau} \to ()/F \quad \Gamma \models e_i : \tau_i, F_i, C_i}{\Gamma \models \texttt{thread}\ A(\overline{e}) : F \cup F_i, \cup C_i \cup NotPriv(\tau_i)}$$

$$\frac{A : \overline{\tau} \to ()/F \quad \Gamma \models e_i : \tau_i, F_i, C_i}{\Gamma \models \texttt{run}\ A(\overline{e}) : F \cup F_i, \cup C_i \cup NotPriv(\tau_i)}$$

$$\Gamma \models \texttt{cooperate} : \emptyset, \emptyset$$

$$\frac{\Gamma \models e : \tau, F_1, C_1 \quad \Gamma \cup x : \tau \models P : F_2, C_2}{\Gamma \models \texttt{let}\ x = e\ \texttt{in}\ P : \cup F_i, \cup C_i}$$

$$\frac{\Gamma \models e_1 : \tau\ \texttt{ref}_\alpha, F_1, C_1 \quad \Gamma \models e_2 : \tau, F_2, C_2}{\Gamma \models e_1 := e_2 : \cup F_i \cup \{\alpha\}, \cup C_i}$$

$$\frac{\Gamma \models e : \tau, F_0, C_0 \quad \Gamma \cup p : \tau \models P_1 : F_1, C_1 \quad \Gamma \models P_2 : F_2, C_2}{\Gamma \models [e \rhd p]P_1, P_2 : \cup F_i, \cup C_i}$$

$$\frac{\Gamma \models e : \mathit{bool}, F_1, C_1 \quad \Gamma \models P : F_2, C_2}{\Gamma \models \texttt{while}\ e\ \texttt{do}\ P : \cup F_i, \cup C_i}$$

Expressions

$$\Gamma \cup x : \tau \models x : \tau, \emptyset, \emptyset$$

$$\frac{f : \overline{\tau} \to \tau'/F \quad \Gamma \models e_i : \tau_i, F_i, C_i}{\Gamma \models f(\overline{e}) : \tau', F \cup F_i, \cup C_i}$$

$$\frac{C : \overline{\tau} \to t/\emptyset \quad \Gamma \models e_i : \tau_i, F_i, C_i}{\Gamma \models C(\overline{e}) : t, \cup F_i, \cup C_i}$$

$$\frac{\Gamma \models e : \tau, F, C}{\Gamma \models \texttt{ref}_\alpha e : \tau\ \texttt{ref}_\beta, F, C \cup \{(\alpha = \beta)\} \cup \mathit{Propagate}(\beta, \tau)}$$

$$\frac{\Gamma \models e : \tau\ \texttt{ref}_\alpha, F, C}{\Gamma \models !e : \tau, F \cup \{\alpha\}, C}$$

7.1 Coherency

Coherence of type inference can be stated as:

Proposition 4 *Suppose that* $\Gamma \models P : F, C$. *If* C *is consistent, then* $\sigma(\Gamma) \vdash \sigma(P) : \sigma(F)$, *for all* σ *unifier of* C.

Proof by induction on the structure of P.

Nothing One has $\Gamma \models \mathbf{0} : \emptyset, \emptyset$. As $\Gamma \vdash \mathbf{0} : \emptyset$, then for all σ, one has $\sigma(\Gamma) \vdash \sigma(\mathbf{0}) : \sigma(\emptyset)$. The same arguments hold for `cooperate`.

Sequence Let us suppose:

$$\frac{\Gamma \models P_i : F_i, C_i}{\Gamma \models P_1; P_2 : \cup F_i, \cup C_i}$$

with $\cup C_i$ coherent and σ a unifier of $\cup C_i$. By induction, as each C_i is coherent and as σ is a unifier of each C_i, one has $\sigma(\Gamma) \vdash \sigma(P_i) : \sigma(F_i)$. This implies $\sigma(\Gamma) \vdash \sigma(P_1; P_2) : \sigma(F_1 \cup F_2)$ because $\sigma(P_1; P_2) = \sigma(P_1); \sigma(P_2)$ and $\sigma(F_1 \cup F_2) = \sigma(F_1) \cup \sigma(F_2)$.

Very similar arguments hold for `while`, `match`, and for function and constructor calls.

Let Let us suppose:

$$\frac{\Gamma \models e : \tau, F_1, C_1 \quad \Gamma \cup x : \tau \models P : F_2, C_2}{\Gamma \models \texttt{let}\, x = e\, \texttt{in}\, P : \cup F_i, \cup C_i}$$

with $\cup C_i$ coherent and σ a unifier of $\cup C_i$. By induction, one has $\sigma(\Gamma) \vdash \sigma(e) : \sigma(\tau), \sigma(F_1)$ and $\sigma(\Gamma \cup x : \tau) \vdash \sigma(P) : \sigma(F_2)$. Then, one has $\sigma(\Gamma) \cup x : \sigma(\tau) \vdash \sigma(P) : \sigma(F_2)$, which implies the result $\sigma(\Gamma) \vdash \texttt{let}\, x = \sigma(e)\, \texttt{in}\, \sigma(P) : \sigma(F_1 \cup F_2)$.

Dereferencing Let us suppose:

$$\frac{\Gamma \models e : \tau\ \texttt{ref}_\alpha, F, C}{\Gamma \models !e : \tau, F \cup \{\alpha\}, C}$$

with C coherent and σ a unifier of C. By induction, one has $\sigma(\Gamma) \vdash \sigma(e) : \sigma(\tau\ \texttt{ref}_\alpha), \sigma(F)$. Let $n = \sigma(\alpha)$. One has $\sigma(\Gamma) \vdash \sigma(e) : \sigma(\tau)\ \texttt{ref}_n, \sigma(F)$. This implies $\sigma(\Gamma) \vdash !\sigma(e) : \sigma(\tau), \sigma(F) \cup \{n\}$ and finally $\sigma(\Gamma) \vdash \sigma(!e) : \sigma(\tau), \sigma(F \cup \{\alpha\})$.

Very similar arguments hold for assignment.

Migration Let us suppose:

$$\frac{\Gamma \models P : F, C}{\Gamma \models P@n : \emptyset, C \cup \{(\alpha \leq n) \,/\, \alpha \in F\}}$$

Let $C' = C \cup \{(\alpha \leq n) \,/\, \alpha \in F\}$ and suppose C' coherent and σ is a unifier of C'. By induction, one has $\sigma(\Gamma) \vdash \sigma(P) : \sigma(F)$ because C is coherent and σ is also a unifier of C. For all $\alpha \in F$, one has $\sigma(\alpha) \leq n$, thus $\sigma(F) \subseteq \{\_, n\}$, which implies the result $\sigma(\Gamma) \vdash \sigma(P@n) : \sigma(\emptyset)$.

Reference Creation Let us suppose:

$$\frac{\Gamma \models e : \tau, F, C}{\Gamma \models \texttt{ref}_\alpha e : \tau\ \texttt{ref}_\beta, F, C \cup \{(\alpha = \beta)\} \cup Propagate(\beta, \tau)}$$

Let $C' = C \cup \{(\alpha = \beta)\} \cup Propagate(\beta, \tau)$ and suppose C' coherent and σ an unifier of C'. Let $n = \sigma(\alpha)$.

By induction, one has $\sigma(\Gamma) \vdash \sigma(e) : \sigma(\tau), \sigma(F)$ because C is coherent and σ is also a unifier of C.

Let us suppose $n = \_$. Then, one has $\sigma(\Gamma) \vdash \texttt{ref}_n \sigma(e) : \sigma(\tau)\ \texttt{ref}_n, \sigma(F)$, and thus $\sigma(\Gamma) \vdash \sigma(\texttt{ref}_\alpha e) : \sigma(\tau\ \texttt{ref}_\beta), \sigma(F)$ because, as C' is coherent, $n = \sigma(\beta)$.

Let us now consider the other case, where $n = l$. In addition, one must show that $public(\sigma(\tau))$ which means that $\sigma(\tau)$ is not of the form $\tau'\texttt{ref}\_$. Let us suppose that this is not the case and that one has $\sigma(\tau) = \tau'\texttt{ref}\_$. Then, $Propagate(\beta, \tau)$ implies $\sigma(\beta) = \_$ which contradicts $n = \sigma(\alpha) = \sigma(\beta) = l$. Thus, one has $public(\sigma(\tau))$ which entails the result.

Thread Creation Let us suppose:

$$\frac{A : \overline{\tau} \to ()/F_0 \quad \Gamma \models e_i : \tau_i, F_i, C_i}{\Gamma \models \texttt{thread}\ A(\overline{e}) : \cup F_i, \cup C_i \cup NotPriv(\tau_i)}$$

Let $C' = \cup C_i \cup NotPriv(\tau_i)$ and suppose C' coherent and σ a unifier of C'. By induction, one has, for all i, $\sigma(\Gamma) \vdash \sigma(e_i) : \sigma(\tau_i), \sigma(F_i)$.

Let us prove that $public(\sigma(\tau_i))$. If $\sigma(\tau_i)$ is not of the form $\tau'\texttt{ref}_n$ then this is true by definition of *public*. So, suppose $\sigma(\tau_i) = \tau'\texttt{ref}_n$. Then there exists α and τ'' such that $\sigma(\alpha) = n$ and $\tau_i = \tau''\texttt{ref}_\alpha$. As σ unifies $NotPriv(\tau_i)$, $n = \sigma(\alpha) \neq \_$, and thus $public(\sigma(\tau_i))$. In both cases, one has $\sigma(\Gamma) \vdash \texttt{thread}\ A(\sigma(\overline{e})) : \cup\sigma(F_i)$.

The `run` instruction is treated in exactly the same way.

□

7.2 Minimality

One now considers the set $Unif$ of unifiers of a set C of constraints infered from a program P. Each unifier $\sigma \in Unif$ is a function that assigns in a consistent way a scheduler name, l or $\_$, to each scheduler variable α appearing in C. One orders elements of $Unif$ by pointwise extension of the relation $\leq$ defined by: $n_1 \leq n_2$ if $n_1 = \_$ or $n_1 = n_2$.

Proposition 5 *If $Unif \neq \emptyset$ then it has a minimal element.*

C and the number of scheduler names appearing in P are finite, so $Unif$ is also finite. One build the minimal element, by considering in turn the scheduler variables $\alpha_1, \ldots, \alpha_n$ appearing in C.

Step 0: Let *result* the function such that $\forall \alpha_i . result(\alpha_i) = \_$, and let $index = 0$.

Step 1: if $index = n$ then goto Step 3. Otherwise, increment $index := index + 1$, then search for an element $\sigma \in Unif$ such that $\sigma(\alpha_{index}) = l$. If no such σ can be found, then goto Step 1. Else, consider $\sigma' = \sigma[\alpha_{index} \backslash \_]$. If $\sigma' \in Unif$, then goto Step 1. Otherwise, change $result$ by $result := result[\alpha_{index} \backslash l]$ and goto Step 1.

Step 3: Return $result$ as the minimal element of $Unif$.

Correction of the algorithm relies on the following:

Proposition 6 *Consider $\sigma \in Unif$ and suppose $\sigma(\alpha) = l$ for some α in C. Let $\sigma' = \sigma[\alpha \backslash \_]$. If $\sigma' \notin Unif$, then $\forall \gamma \in Unif. \gamma(\alpha) = l$.*

Indeed, $\sigma(\alpha) = l$ comes from an application of the following rule (this is the only way to introduce explicit scheduler names of the form l in constraints):

$$\frac{\Gamma \models P' : F, C'}{\Gamma \models P'@l : \emptyset, C' \cup \{(\alpha \leq l) \,/\, \alpha \in F\}}$$

with $C = C' \cup \{(\alpha \leq l) \,/\, \alpha \in F\}$. By recurrence, let θ the minimal unifier of C' infered from P'. If $\theta(\alpha) = \_$, then θ is also the minimal unifier for C. If $\theta(\alpha) = l'$, then by recurrence all unifiers of C', and hence all unifiers of C, assign to α the same value, which leads to $l' = l$.
□

8 Resource Control

The goal is now to control the size of the heap used by a program. There are actually two sub-goals: first, to control the size of the data stored in references; second, to control the number of references accessible by the program. Here, to control basically means to get a bound depending only on the size of the program input. To simplify, one considers that the program input is initially stored in the heap.

One assigns to each reference a *level* which is an integer (levels will always appear as superscripts). The syntax of expressions becomes:

$e ::= x \mid r^k \mid f(\overline{e}) \mid C(\overline{e}) \mid \texttt{ref}\ ^k e \mid\ !e$

In the reference value r^k, k is the level of r. A level is also associated to reference creations: all references created by evaluation of $\texttt{ref}\ ^k e$ have level k. The rule for reference creation becomes:

$$\frac{e, \mathcal{H} \Downarrow v, \mathcal{H}' \quad r^k \notin Dom(\mathcal{H}')}{\texttt{ref}\ ^k e, \mathcal{H} \Downarrow r^k, \mathcal{H}' \oplus [r^k \backslash v]}$$

8.1 Type System for Stratification

Type judgments now have the form $\Gamma \vdash P : R, W, \mathcal{G}$ for programs, and the form $\Gamma \vdash e : \tau, R, W, \mathcal{G}$ for expression. In these forms, R is the set of levels of possibly read references, W is the set of levels of possibly written references, and $\mathcal{G}$ is the graph of reference accesses. In $\mathcal{G}$, nodes are levels and an arc from r to w means that the content of a reference of level r is possibly transfered to a reference of level w.

The notation $Dep(A, B)$ denotes the graph whose set of nodes is $A \cup B$ and in which there is an arc from each element of B to each element of A.

Programs

$$\Gamma \vdash \mathbf{0} : \emptyset, \emptyset, \emptyset$$

$$\frac{\Gamma \vdash P : R_1, W_1, \mathcal{G}_1 \quad \Gamma \vdash e : \mathit{bool}, R_2, W_2, \mathcal{G}_2,}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ P : \cup R_i, \cup W_i, \cup \mathcal{G}_i}$$

$$\frac{\Gamma \vdash P_i : R_i, W_i, \mathcal{G}_i}{\Gamma \vdash P_1; P_2 : \cup R_i, \cup W_i, \cup \mathcal{G}_i}$$

$$\frac{\Gamma \vdash P : R, W, \mathcal{G}}{\Gamma \vdash P@n : R, W, \mathcal{G}}$$

$$\frac{A : \overline{\tau} \to ()/R, W \quad \Gamma \vdash e_i : \tau_i, R_i, W_i, \mathcal{G}_i}{\Gamma \vdash \texttt{thread}\ A(\overline{e}) : R \cup R_i, W \cup W_i, \cup \mathcal{G}_i \cup \mathit{Dep}(W, \cup R_i)}$$

$$\frac{A : \overline{\tau} \to ()/R, W \quad \Gamma \vdash e_i : \tau_i, R_i, W_i, \mathcal{G}_i}{\Gamma \vdash \texttt{run}\ A(\overline{e}) : R \cup R_i, W \cup W_i, \cup \mathcal{G}_i \cup \mathit{Dep}(W, \cup R_i)}$$

$$\Gamma \vdash \texttt{cooperate} : \emptyset, \emptyset, \emptyset$$

$$\frac{\Gamma \vdash e : \tau, R_1, W_1, \mathcal{G}_1 \quad \Gamma \cup x : \tau \vdash P : R_2, W_2, \mathcal{G}_2}{\Gamma \vdash \texttt{let}\ x = e\ \texttt{in}\ P : \cup R_i, \cup W_i, \cup \mathcal{G}_i \cup \mathit{Dep}(W_2, R_1)}$$

$$\frac{\Gamma \vdash e_1 : \tau\ \texttt{ref}^k, R_1, W_1, \mathcal{G}_1 \quad \Gamma \vdash e_2 : \tau, R_2, W_2, \mathcal{G}_2}{\Gamma \vdash e_1 := e_2 : \cup R_i, \cup W_i \cup \{k\}, \cup \mathcal{G}_i \cup Dep(\{k\}, R_2)}$$

$$\frac{\Gamma \vdash e : \tau, R_0, W_0, \mathcal{G}_0 \quad \Gamma \cup p : \tau \vdash P_1 : R_1, W_1, \mathcal{G}_1 \quad \Gamma \vdash P_2 : R_2, W_2, \mathcal{G}_2}{\Gamma \vdash [e \rhd p]P_1, P_2 : \cup R_i, \cup W_i, \cup \mathcal{G}_i}$$

Expressions

$$\Gamma \cup x : \tau \vdash x : \tau, \emptyset, \emptyset, \emptyset$$

$$\frac{f : \overline{\tau} \to \tau'/R, W \quad \Gamma \vdash e_i : \tau_i, R_i, W_i, \mathcal{G}_i}{\Gamma \vdash f(\overline{e}) : \tau', R \cup R_i, W \cup W_i, \cup \mathcal{G}_i \cup Dep(W, \cup R_i)}$$

$$\frac{C : \overline{\tau} \to t/\emptyset, \emptyset \quad \Gamma \vdash e_i : \tau_i, R_i, W_i, \mathcal{G}_i}{\Gamma \vdash C(\overline{e}) : t, \cup R_i, \cup W_i, \cup \mathcal{G}_i}$$

$$\frac{\Gamma \vdash e : \tau, R, W, \mathcal{G}}{\Gamma \vdash \texttt{ref}^k e : \tau\ \texttt{ref}^k, R, W, \mathcal{G}}$$

$$\frac{\Gamma \vdash e : \tau\ \texttt{ref}^k, R, W, \mathcal{G}}{\Gamma \vdash !e : \tau, R \cup \{k\}, W, \mathcal{G}}$$

Stratification Let P a well-typed program such that $\emptyset \vdash P : R, W, \mathcal{G}$. One says that P is *stratified* if $\mathcal{G}$ is acyclic. In this case, levels can be partitionned in strates.

For each value v of type τ one defines the size of v in the heap $\mathcal{H}$, noted $|v|_{\mathcal{H}}$, by:

- If τ is a basic type (unit, int, bool,...), then $|v|_{\mathcal{H}} = 1$.
- If $\tau = \tau'\ \texttt{ref}$, then $|v|_{\mathcal{H}} = 1 + |\mathcal{H}(v)|_{\mathcal{H}}$.
- If $\tau = t$ an inductive type and $v = C(v_0, \ldots, v_n)$, then $|v|_{\mathcal{H}} = 1 + |\mathcal{H}(v_0)|_{\mathcal{H}} + \cdots + |\mathcal{H}(v_n)|_{\mathcal{H}}$.

One notes $max(\mathcal{H})$ the maximum of the sizes of all the data stored in $\mathcal{H}$: $max(\mathcal{H}) = max\{|\mathcal{H}(r)|_{\mathcal{H}} / r \in Dom(\mathcal{H})\}$.

The size of references created by a stratified program is bounded and the bound only depends on the size of the program input:

Proposition 7 *Let P a stratified program. Then, there exists a function f_P such that, whenever $P, \mathcal{H} \rightarrow P_1, \mathcal{H}_1 \rightarrow P_2, \mathcal{H}_2 \rightarrow \cdots$, then $max(\mathcal{H}_i) < f_P(max(\mathcal{H}))$ for all i.*

8.2 Type System for Living Threads

One defines a type system to insure that the number of living threads remains bounded. Actually, one only needs to avoid asynchronous thread creation in loops. The system is very simple:

$$\mathbf{0} \triangleright \mathbf{false}$$

$$e_1 := e_2 \triangleright \mathbf{false}$$

$$\texttt{thread}\ A(\overline{e}) \triangleright \mathbf{true}$$

$$\texttt{cooperate} \triangleright \mathbf{false}$$

$$\texttt{run}\ A(\overline{e}) \triangleright \mathbf{false}$$

$$\frac{P_1 \triangleright b_1 \quad P_2 \triangleright b_2}{P_1; P_2 \triangleright b_1 \vee b_2}$$

$$\frac{P_1 \triangleright b_1 \quad P_2 \triangleright b_2}{[e \triangleright p]P_1, P_2 \triangleright b_1 \vee b_2}$$

$$\frac{P \triangleright b}{\texttt{let}\ x = e\ \texttt{in}\ P \triangleright b}$$

$$\frac{P \triangleright b}{P@n \triangleright b}$$

$$\frac{P \triangleright \mathbf{false}}{\texttt{while}\ e\ \texttt{do}\ P \triangleright \mathbf{false}}$$

The last rule rejects loops whose body can create an asynchronous thread.

Proposition 8 *Let P be a stratified program such that $P \triangleright b$. Then, the number of living threads is bounded and the bound only depends on the size of the program input.*

8.3 Bounded Memory

References are dynamically created when an expression of the form `ref` e is evaluated. Thus, the number of created references can be unbound, as for example in the following program where a reference is created at each instant:

```
while true do begin ref 0; cooperate end
```

However, the number of references accessible by a given thread remains finite and bound by a value which can be extracted from the syntax (the number of `let` instructions). For example, in the previous example, no reference is actually accessible by the executing thread (references can be immediately garbage collected, as soon as they are created).

Thus, in a stratified program, the number of living threads, the number of references accessible by each thread, and the size of each reference are all bound by a value depending on the program input. As a consequence, the total size of the heap (the sum of the sizes of all accessible references) is bound by a value depending on the program input, provided the program is stratified.

Proposition 9 *Let P be a stratified program such that $P \triangleright b$. Then, the memory used by P is bounded and the bound only depends on the size of the program input.*

9 Detection of Function Termination

FunLoft allows users to define functions. However, the termination of functions is tested and mandatory. Indeed, a linked thread calling a non-terminating function would get stuck executing it and it would prevent the other threads linked to the same scheduler to get the control. For that purpose, loops are forbidden and recursivity is controlled. More precisely, the syntax of function bodies is:

$F ::= x \mid r \mid f(\overline{e}) \mid C(\overline{e}) \mid \texttt{ref}\, e \mid !e \mid F; F \mid \texttt{let}\, x = e\, \texttt{in}\, F \mid e := e \mid [e \triangleright p]F, F$

An example of function definition is:

$length(x) = [x \triangleright Cons(h,t)]plus\_int(1, length(t)), 0$

which in real syntax should be written:

```
let length (x) =
   match x with Cons_list (h,t) -> 1+length (t) | default -> 0
```

The definition of the function length is correct; indeed, there is only one sequence of calls starting from the initial call `length (x)` and reaching the call `length (t)`. As `x` matches `I_cons (h,t)`, `t` is a strict sub-term of `x`. Thus, no infinite sequence of calls of `length` can exists, as the size of the parameter decreases at each call.

More precisely, one says that the size of a parameter `p` is smaller than the size of a parameter `q` if `p` is a sub-term of `q`. For lists of parameters, one extends "lexicographically" the notion of size. Finally, in a complete sequence of calls (starting and ending on the same function), one checks that at each

step parameters are strictly decreasing. Thus, any function call is forced to terminate after a finite number of recursive calls.

In the present version of FunLoft, recursivity can only concern parameters of inductive types. The following definition, in which recursivity concerns a parameter of integer type, is thus rejected:

```
let fact (n) =
   if n = 0 then 1 else n*fact (n-1)
```

Note that detection of termination of functions is an independent issue which does not interfere with previous issues such as the control over the number of simultaneously living threads. This does not remain true when creation of threads by functions is allowed; for simplicity, we do not consider this feature here.

Detection of termination of functions, as presently done in FunLoft, is a topics that should certainly be improved in further versions of the language. For example, in a sequence of calls leading from a function definition to a call of the same function, one could accept that some calls keep the parameters unchanged, provided there exists at least one call that strictly decreases the parameters size.

10 Events

One now introduces events which are used both for thread synchronisation and for thread communication. Previous syntax for expressions and programs is extended by:

e ::=
   ... | `event` | s | `pre` s

P ::=
   ... | `generate` e, e | `await` e | `get_all` e `in` e

- `event` declares a new fresh event. Events are denoted by s.
- `pre` s is the list of values generated with S during previous instant.
- `generate` e_1, e_2 generates the event denoted by e_1 and adds the value denoted by e_2 to the list of current values of e_1.
- `await` e blocks the control until the event denoted by e is generated.
- `get_all` e_1 `in` e_2 blocks the control during the current instant and collects during it the values generated for the event denoted by e_1. The list of collected values is assigned to the reference denoted by e_2 and become available at next instant.

10.1 Operational Semantics

An environment of events maps events to list of values. One considers couples of environments of events $\mathcal{E}_c, \mathcal{E}_p$, where $\mathcal{E}_c(s)$ is the list of values generated for the event s during the current instant, and $\mathcal{E}_p(s)$ stores the list of values that have been generated for s during the previous instant. One notes $\emptyset$ the environment undefined for each event, and Nil_list the empty list.

One defines a predicate which tests in an environment the absence of value associated to a givent event. This predicate will be used to test if an event has already been generated during the current instant: $present(s, \mathcal{E}) \Leftrightarrow \mathcal{E}(s) \neq Nil\_list$.

Event Creation An event creation extends the event environment with a new fresh location:

$$\frac{s \notin Dom(\mathcal{E})}{\texttt{event}, \mathcal{H}, \mathcal{E}_c, \mathcal{E}_p \ \Downarrow \ s, \mathcal{H}, \mathcal{E}_c \oplus [s \backslash Nil\_list], \mathcal{E}_p}$$

Evaluation of an event returns it:

$$s, \mathcal{H}, \mathcal{E}_c, \mathcal{E}_p \ \Downarrow \ s, \mathcal{H}, \mathcal{E}_c, \mathcal{E}_p$$

The expression `pre` s returns the list of values generated with s during the previous instant (the empty list, if there was no generation at all).

$$\texttt{pre}\ s, \mathcal{H}, \mathcal{E}_c, \mathcal{E}_p \ \Downarrow \ \mathcal{E}_p(s), \mathcal{H}, \mathcal{E}_c, \mathcal{E}_p$$

Event Generation The generation of an event adds the value generated to the list of current values:

$$\frac{(e_1, e_2), \mathcal{H}, \mathcal{E}_c, \mathcal{E}_p \ \Downarrow \ (s, v), \mathcal{H}', \mathcal{E}'_c, \mathcal{E}'_p}{\texttt{generate}\ e_1, e_2, \mathcal{H}, \mathcal{E}_c, \mathcal{E}_p \rightarrow \mathbf{0}, \mathcal{H}', \mathcal{E}'_c[s \backslash Cons\_list(v, \mathcal{E}'_c(s))], \mathcal{E}'_p}$$

Await Awaiting an event which is not present (not previously generated during the current instant) transfers the executing thread in the waiting list:

$$\frac{e, \mathcal{H}, \mathcal{E}_c, \mathcal{E}_c \ \Downarrow \ s, \mathcal{H}', \mathcal{E}'_c, \mathcal{E}'_c \quad \neg present(s, \mathcal{E}'_c)}{C[< E[\texttt{await}\ e]_t \cdot L, L' >_l], \mathcal{H}, \mathcal{E}_c, \mathcal{E}_p \rightarrow C[< L, L' \cdot E[\texttt{await}\ s]_t >_l], \mathcal{H}', \mathcal{E}'_c, \mathcal{E}_p}$$

Awaiting an already generated event has no effect:

$$\frac{e, \mathcal{H}, \mathcal{E}_c, \mathcal{E}_p \ \Downarrow \ s, \mathcal{H}', \mathcal{E}'_c, \mathcal{E}'_p \quad present(s, \mathcal{E}'_c)}{\texttt{await}\ e, \mathcal{H}, \mathcal{E}_c, \mathcal{E}_p \rightarrow \mathbf{0}, \mathcal{H}', \mathcal{E}'_c, \mathcal{E}'_p}$$

It is possible to resume execution of a waiting thread if the awaited event is present. This rule is used to resume execution of threads waiting for an event which becomes generated.

$$\frac{present(s, \mathcal{E}_c)}{C[< L, L' \cdot E[\texttt{await}\ s]_t \cdot L'' >_l], \mathcal{T}, \mathcal{H}, \mathcal{E}_c, \mathcal{E}_p \rightarrow C[< L \cdot E[\mathbf{0}]_t, L' \cdot L'' >_l], \mathcal{T}, \mathcal{H}, \mathcal{E}_c, \mathcal{E}_p}$$

Get all values The way to get the values of an event generated during the previous instant is in FunLoft to use the `get_all_values` which is actually expressible through the `pre` operator:

$$\frac{(e_1, e_2), \mathcal{H}, \mathcal{E}_c, \mathcal{E}_p \ \Downarrow\ (s, r), \mathcal{H}', \mathcal{E}'_c, \mathcal{E}'_p}{C[< E[\texttt{get\_all}\ e_1\ \texttt{in}\ e_2]_t \cdot L, L' >_l], \mathcal{T}, \mathcal{H}, \mathcal{E}_c, \mathcal{E}_p \rightarrow C[< L, L' \cdot E[r := \texttt{pre}\ s]_t >_l], \mathcal{T}, \mathcal{H}', \mathcal{E}'_c, \mathcal{E}'_p}$$

Termination Relation One defines a predicate to test if there is no thread waiting for a present event. The $\downarrow^{\mathcal{E}}_{term}$ predicate is defined by:

$$\downarrow^{\mathcal{E}}_{term} S = \texttt{false} \Leftrightarrow S =< L, L' \cdot E[\texttt{await}\ s] \cdot L'' >_n \ \ \wedge\ present(s, \mathcal{E})$$

The end of instant is now defined as the moment where no thread remains to be executed nor is bloqued on a present event. At the end of each instant, the current environment is saved in the previous environment, before being reset:

$$\frac{S_i \downarrow^{\mathcal{E}_c}_{term} \quad S_i \downarrow^{\mathcal{T}}_{eoi} S'_i \quad S_i \in \widehat{S_1} = \{S_1, \ldots, S_k\}}{C[S_1] \ldots [S_k], \mathcal{T}, \mathcal{H}, \mathcal{E}_c, \mathcal{E}_p \rightarrow C[S'_1] \ldots [S'_k], \mathcal{T}, \mathcal{H}, \emptyset, \mathcal{E}_c}$$

10.2 Stratification

Events should participate to stratification as well as references. For example, the following module should be rejected as it violates stratification:

```
let head (l) =
  match l with Nil_list -> Nil_list | Cons_list (a,m) -> a end

let module m (e) =
  let r = ref Nil_list in
     loop
        begin
           generate e with Cons_list (0,head (!r));
           get_all_values e in r;
        end
```

By accepting it, one would cyclically store in the reference `r` a list of lists with increasing size, which is clearly a memory leak.

10.3 Communication Between Schedulers

In FunLoft, an event is associated to one unique scheduler, or to an area of synchronised schedulers. The case of an unique scheduler will be considered as a special case of an area with only one element. There should be impossible to remotely generate events when schedulers are not synchronised:

```
let s1 = scheduler
let s2 = scheduler
let e = event

let module wait () = link s1 do await e

let module gen () =
  let v = ref 0 in
    link s2 do
       loop begin
          generate e with v;
          v++;
          cooperate;
       end
```

Indeed, the number of generations to be stored is not bounded (it actually depends on the relative speed of the two schedulers) which is a potential source of memory leak.

However, use of events becomes possible between synchronised schedulers because the schedulers share the same instants; the following program is thus correct:

```
let s1 = scheduler
and s2 = scheduler
let e = event
let module wait () = link s1 do await e
let module gen () = link s2 do generate e
```

10.4 Choice of Primitives

The `pre` operator is powerful enough to express the `get_all_values` primitive of FunLoft. Indeed, the instruction `get_all_values s in r` is equivalent to the sequence `begin cooperate; r:=pre s end`.

However, there is a difference at implementation level: without an analyse to determine the absence of execution of `pre`, the list of values generated during an instant must be stored in order to be available at the next instant. This is not the case with the primitive `get_all_values`: the list of previously generated values needs to be constructed only when the primitive is executed.

Actually, FunLoft also proposes the primitive `for_all_values` with the syntax `for_all_values s with x -> e` interpreted as: for each value generated for event `s` during the current instant, evaluate the expression `e`, in which the variable `x` is bound to the generated value. Thus, the call-back `e` is immediately

evaluated during the current instant for each generation, as soon as it occurs. Like with `get_all_values`, one has to wait for next instant for termination, in order to be sure that all generations have been processed, but now reactions to generations are immediate. Two points are to be noted:

- Implementation is made easier, as no list of generated values is built.
- Stratification forbids the implementation of `get_all_values` using the primitive `for_all_values` as the number of values generated during an instant for a given event is not statically known.

11 Complete Language

Several features of FunLoft are missing from the present formalisation:

- Possibility for functions to create new (asynchronous) threads and to generate events.
- Finite loops (`repeat`), and infinite loops (`loop`).
- Arrays of references.
- Instruction to force a thread to definitively terminate (also, to suspend/resume a thread).
- Embedding of references in inductive data type.

We are now considering these features in turn.

Threads Created by Function Asynchronous thread creations are FunLoft expressions. The type system of 8.2 should thus be extended to expressions, to trace asynchronous thread creation (actually, this is implemented as a supplementary effect added to the standard type system). For example, in the following code, the type information produced for function `f` indicates an effect of thread creation, which is used to reject the definition of the module `m1`:

```
let f () = thread m0 ()
let module m1 () =
   while true do
      begin
         f ();
         cooperate;
      end
```

Note that one gets a new relation between thread creation and data processing. For example, consider:

```
let nat = Z | S of nat
let f (n) =
   match n with
     S (m) -> begin thread m0 (); f (m); end
   | default -> ()
```

The number of threads created depends on the parameter of the function `f`. As the size of the parameter is bounded, the number of created threads is also bounded.

Events Generated by Function In FunLoft, event generations are considered as expressions. Events can therefore be generated by functions. This has two consequences:

- One has to define a new effect denoting event generation. This effect is associated to functions that generate events. It will be used for stratification checking (see 10.2).
- In `for_all_values` (see 10.4) one has to verify the absence of effect denoting an event generation in the call-back expression. Indeed, a cyclic evaluation could occur if the evaluation of a call-back would lead to the re-generation of the awaited event, as in `for_all_values s with x do generate s`.

Finite Loops FunLoft introduces finite loops under the form of `repeat` instructions. Finite loops can occur both in functions and in modules. They cannot introduce cyclic behaviours, and they terminate instantly, provided their body also terminate instantly. The operational semantics (for simplicity, events are not considered) is:

$$\frac{e, \mathcal{H} \Downarrow n, \mathcal{H}' \quad n \leq 0}{\texttt{repeat}\ e\ \texttt{do}\ P, \mathcal{H} \rightarrow \mathbf{0}, \mathcal{H}'}$$

$$\frac{e, \mathcal{H} \Downarrow n, \mathcal{H}' \quad n \geq 1 \quad P, \mathcal{H} \rightarrow P', \mathcal{H}'}{\texttt{repeat}\ e\ \texttt{do}\ P, \mathcal{H} \rightarrow P'; \texttt{repeat}\ n-1\ \texttt{do}\ P, \mathcal{H}'}$$

In `repeat e do P`, the expression `e` needs not to be static (computable at compile time).

The factorial function considered in section 9 can be coded using the `repeat` primitive:

```
let fact (n) =
  let res = ref 1 in
  let count = ref n in
  begin
```

```
    repeat n do begin
      res:=!res*!i;
      i--
    end;
    !res
  end
```

Note that stratification is not violated because the type of the values stored in the reference `res` is not inductive (it's `int`).

Infinite Loops Infinite loops never terminate. Actually, the infinite loop `loop p` is operationally equivalent to `while true do p`. By defining an infinite loop, the programmer tells the system that the loop never terminates, and this information can be used by the type system to detect instantaneous loops.

The typing of infinite loops in the system of section 4 is thus:

$$\frac{P \Vdash \mathbf{false}}{\texttt{loop}\, P \Vdash \mathbf{false}}$$

Arrays Arrays of references are definable in FunLoft. The syntax is `ref[n] v` which creates an array of `n` references all intialised with `v`. The expression `n` needs not to be static. Elements of arrays are usually processed in turn with the aid of finite loops.

The element of index `i` in an array `a` is noted `a[i]`. Indexing is made modulo the array size; this forbids run-time out-of-bounds errors. Thus, in FunLoft, arrays are basically circular buffers.

Arrays are treated exactly as if they were single references for stratification and elimination of data-races. For example, the following code is rejected, despite the fact that no data-race actually occurs:

```
let s1 = scheduler
let s2 = scheduler
let array = ref [2] 0
let module first () = link s1 do array[0]:=1
let module second () = link s2 do array[1]:=2
```

The type system is not able to detect the disjointness of the two parts of the array accessed by the two modules. Note that the program is also rejected if the two schedulers are synchronised.

Controlling Threads In FunLoft there is the possibility to kill a thread, by executing a `stop` instruction. A thread can be also suspended and resumed, with the `suspend`, `resume` instructions. These instructions have no effect on unlinked threads. To formalise these instructions does not present any difficulty and is left to the reader.

References in Inductive Types References can be embedded in user-defined types, as in:

```
type bullet_t =
  Bullet of
      int ref   *    // x coord
      int ref   *    // y coord
      int ref   *    // x speed
      int ref        // y speed
```

Inductive types can be recursively defined, as in:

```
type rlist = Nil_rlist | Cons_rlist of int ref * rlist
```

which defines lists of integer references.

The type system for atomicity described in section 5 must be adapted to take in account the possibility to embedd references into types. Basically, one needs to change the definition of a public type: a user-defined type is public iff it does not embedd any private reference and if it is build only from public types.

Polymorphism FunLoft allows polymorphism both at data type level and at function level. Fo example, the following function defines the polymorphic identity: `let id (x) = x`. Type inference has to be performed in presence of polymorphism. For example, the function `id` receives type $\forall\alpha.\alpha \to \alpha$.

At data level, types can have parameters, as in:

```
type 'a list = Nil_list | Cons_list of 'a * list
```

which defines the list of elements of the parametric type `'a`.

As usual in presence of polymorphism (in OCaml[3], for example), one has to control *variable generalisation*. More precisely, if the system assigns to a variable an incomplete parametric reference, array, or event type (i.e. a type in which parameters remains), then this variable should not be used in contexts where the parameter receives several distinct types. One says that such a variable is *not generalisable*. For example, consider:

```
type 'a cell_data = Undef | Cell of 'a
let x = ref Undef
let f (v) =
 let z = !x in
     match z with Cell (c) -> x := Cell (v) | default -> ()
```

The type of `x` is not complete (one does not know what is `'a`) and `x` is thus not generalisable. The function `f` has an effect which is to assign to `x` a value given in parameter. The following function is erroneous as it first sets the parameter `'a` of `x` to the boolean type, then to the integer type:

```
let g () =
   begin
      f (true);
      f (1);
   end
```

The problem is similar with events, and the following program is also rejected (`e` is not generalisable, but used in two distinct contexts):

```
let e = event
let f1 (x) = generate e with x
let g () =
  begin
    f1 (true)
    f1 (1);
  end
```

12 Related Work

Synchronous Reactive Programming Considering concurrency, FunLoft finds its roots in reactive programming[2] which is itself issued from the activity developped around synchronous languages[7].

Synchronous languages, such as Esterel[8] and Lustre[12], put the focus on the validation of programs by means of formal methods. They consider static systems in which dynamic creation of concurrent entities is not allowed. The synchronous language Lucid Synchrone[22] is an exception because it allows recursive definitions.

Synchronous languages are generally build on top of a "base-language", in which are defined data and elementary processings on them. In Esterel, for example, the base-language is C, while in Lucid Synchrone this is Ocaml. Considering resource control, synchronous languages provide means (for example, the "clock-calculus" of Lustre) to assure that execution will run in bounded memory, *provided actions performed in the base-language are proved to run in bounded memory*. Thus, unlike FunLoft, no synchronous language gives complete (that is, including the base-language) ways to control resources.

The ReactiveML[19] library introduces reactive programming in Objective Caml. ReactiveML is as safe as ML. There is no equivalent of unlinked threads in ReactiveML. Moreover, all concurrent activities defined in ReactiveML share the same instants which makes the implementation on multi-processors architectures problematic.

Thread Based Frameworks The Gnu-Pth[15] thread library designed for Unix platforms provides cooperative scheduling in the context of POSIX/ANSI-C. Blocking I/O primitives have been rewritten in order to work in a cooperative scheduling. This differs from our proposal which gives users the freedom to safely code some more elaborated behaviors by escaping the cooperating scheduling.

Moreover, Gnu-Pth, being totally cooperative, has difficulties to fully benefit from multi-cores.

The FairThreads model defines a framework to mix cooperative and preemptive threads in C[10], Java[9], or Scheme[24]. This approach is at the basis of our but it is not concerned nor by safety nor by ensuring that the preemptive level does not interfere with the cooperative one, thus suffering of the usual issues encoutered with preemptive threads.

Cyclone The Cyclone[26] language proposes a safe variant of C by limiting the use of pointers. Technically, regions are used in order to control that a pointer is not used outside its definition scope. An extension of Cyclone to multithreading is proposed in [18]. The main difference with our approach is that only preemptive threads are considered, controlled by locks and with possibilities of deadlocks. However, no implementation seems to exist yet for the multithreaded extension of Cyclone.

Atomicity In [16], a static analyse close to a type and effect system is considered; it focuses on the issue of ensuring the atomicity of programs by means of locks in the framework of a language providing preemptive threads. Although our programming approach differs, the notion of singleton type introduced there seems promising to us as it might be a way to consider the dynamic creation of schedulers. In particular, we'll try to combine it with the notion of region local to a piece of code which is considered in type and effect system to ensure that some memory locations do not escape from their initial scopes. However, it is at the moment not clear to us that such a method would apply as well as the one we use for separating private areas here. Indeed, when using external function, the notion of scope associated to a piece of code might not be as expressive as the notion of scope associated with a thread that we are currently using.

More recently, the problem of atomicity has also been considered for Java methods in [17].

A first version of the type system for atomicity described here for FunLoft has been presented in [14].

Separation Logics Separation Logic[23, 21] offers means to ensure the disjointness of several parts of the memory which might be safely accessed concurrently. It is not clear to us, however, how it could apply to FunLoft, which allows dynamic allocation of memory locations and dynamic creation of threads.

Synchronous-π-calculus Recently, has been proposed the Synchronous-π-calculus[4] (Sπ-calculus) which considers reactive programming from a process calculus point of view. The Sπ-calculus is a synchronous π-calculus which is based on the SL model[11]. The latter is a relaxation of the Esterel model where the reaction to the absence of a signal within an instant can only happen at the next instant. This is exactly the event model on which FunLoft stands.

The Sπ-calculus basically differs from FunLoft in that references are absent from it.

In [5] is considered the way to control resources for a formalism which is very close to the Sπ-calculus. Actually, *polynomial bounds* are produced, which is much more realistic than unconstrained ones that are considered in FunLoft.

Conditions to get deterministic behaviours in the Synchronous-π-calculus are studied in [6] which proposes a typing system that guarantees determinacy. The accent on determinism is also put in the SHIM formalism[25].

PACT *Partially Cooperative Threads* (PACT) is a formalism very close to FunLoft, introduced in F. Dabrowski's thesis[13]. The type system for atomicity of section 5 is presented and fully proved for PACT. PACT does not makes the difference function/module, and thus allows to recursively define behaviours that do not immediately terminate. In the present version, PACT does not considers synchronised schedulers, nor the `join` primitive of FunLoft.

13 Conclusion

One has first considered the basic formalism, without data, covering thread creation, execution, joining, and migration. Scheduler synchronisation is also treated in this basic fragment. Technically, we have introduced a tree environment to reflect the living state of threads, and an end-of-instant relation to synchronise schedulers.

Then, data and references are introduced. Data and functions to manipulate them are considered at an abstract level, without polymorphism. A type system has been given together with a variant of the operational semantics. A region is associated to each reference. The result is the absence of data-races for well-typed programs of this fragment.

Then, we have considered type inference. The goal was to infer the previous regions. Polymorphism is only present at the region level, not at data-type level. The purpose is thus to infer the regions for programs supposed to be correct at the data-type level. When inference is possible, one can deduce that no data-race occur at execution.

Finally, we have introduced events and show the changes induced in the operational semantics and in the type system.

The split in several fragments has had the advantage to isolate issues. For example, the issue of termination detection of functions is orthogonal to the one of avoiding data-races. Thus, termination detection can be postponed to the processing of functions. An other advantage is to avoid to immediately consider recursive definitions in their full generality; actually, recursive definitions are a real issue only for detection of function termination.

One has a strong result: correct FunLoft programs are proved free from data-races and from memory leaks. They are also safe in the sense that no error

should occur at execution time (provided memory is sufficient).[1]

Two questions seem legitimate at that point:

1. Is the existence of bounds sufficient in practice?
2. What programs are ruled out, and what programs pass the filter of the static analysis?

Bounds In FunLoft, only is proved the *existence* of bounds for the memory used and for the CPU consumed. These bounds depend on the size of the program input. But the infered bounds may be unrealistic. Actually, one can for example get exponential bounds, like in:

```
type tree = L of int | T of tree * tree

let module map_on_leaf (t) =
   match t with
      L (n) -> thread leaf_processing (n)
    | T (t1,t2) -> begin map_on_leaf (t1); map_on_leaf (t2); end
   end

let module tree_processing () =
  let t = ref L (0) in
    while true do
      begin
        t:=get_tree ();
        run map_on_leaf (!t);
      end
```

In this example, the loop stays bloqued on the `run` instruction until all the threads created for processing the leaves of the tree `t` are terminated. At each cycle, the number of created threads depends on the number of leaves of `t`. As the size of `t` (the number of calls of constructors in it) is bounded, this number is also bounded, like is the number of active threads in the system. However, the bound exponentially depends on the size of `t`.

Using the `repeat` instruction can also lead to huge unrealistic bounds. For example, `repeat n do thread m ()` could create N threads, where N is the larger storable integer!

Thus, existence of bounds should be considered just as a first step in direction of a fully satisfactory resource control.

Expressivity Consider a program which stores a list of requests all along the passing of instants, and which starts to process the requests when a triggering event becomes present. The need to store an unbounded number of requests, even if each request is of bounded size, is clearly contradictory with the existence

[1] Safety in FunLoft is also related to the capture of divide-by-zero errors; we do not go in deeper details here.

of a bound on the memory consumed. Such a program is thus not implementable in FunLoft (technically, it should be rejected because of stratification violation). Actually, only finite versions of the program are implementable, in which is defined a maximum of requests that can be stored (for example, in an array). Note however that this maximum can be totally unrealistic (it could be the maximal integer storable in memory; see previous paragraph). A standard solution is to define a static pool of threads, each of them cyclically processing requests. FunLoft does not force the programmer to use such a solution, and allows him to dynamically create threads, one for each request.

The code to implement cellular automata is described in [1] and the distribution of the software contains several examples, among them a prey/predator system in which new preys are cyclically injected as soon as old ones have all been killed.

More examples are to be tried with FunLoft to get a clearer response to the second question. Note however, that one is able in FunLoft to get use at language level of the several cores of a multi-core architecture.

References

[1] FunLoft. http://www.inria.fr/mimosa/rp/FunLoft.

[2] Reactive Programming. http://www.inria.fr/mimosa/rp.

[3] The Objective Caml System. http://www.ocaml.org.

[4] Roberto M. Amadio. A Synchronous pi-Calculus, 2006. available at: http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0606019.

[5] Roberto M. Amadio and Frédéric Dabrowski. Feasible Reactivity for Synchronous Cooperative Threads. In *Workshop on Expressiveness in Concurrency*, pages 33–43, San Francisco, 2006. ENTCS 154(3).

[6] Roberto M. Amadio and Mehdi Dogguy. Determinacy in a Synchronous pi-Calculus, July 2007. Technical Report, Université Paris 7, Laboratoire PPS (Submitted).

[7] Albert Benveniste and Gérard Berry. The Synchronous Approach to Reactive and Real-Time System. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[8] Gérard Berry and George Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[9] Frédéric Boussinot. *Java Fair Threads*. Inria research report, RR-4139, 2001.

[10] Frédéric Boussinot. FairThreads: Mixing Cooperative and Preemptive Threads in C. *Concurrency and Computation: Practice and Experience*, vol 18 pp 445-469, 2006.

[11] Frédéric Boussinot and Robert de Simone. The SL Synchronous Language. *IEEE Trans. Softw. Eng.*, 22(4):256–266, 1996.

[12] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A Declarative Language for Programming Synchronous Systems. In *POPL*, pages 178–188, 1987.

[13] Frédéric Dabrowski. *Programmation réactive synchrone - Langage et contrôle des ressources*. PhD thesis, Université Paris 7, 2007.

[14] Frédéric Dabrowski and Frédéric Boussinot. Cooperative Threads and Preemptive Computations. *Proceedings of TV'06, Multithreading in Hardware and Software: Formal Approaches to Design and Verification, Seattle*, 2006.

[15] Ralf S. Engelschall. *Portable Multithreading*. Proc. USENIX Annual Technical Conference, San Diego, California, 2000.

[16] Cormac Flanagan and Martin Abadi. Types for Safe Locking. In *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 91–108, London, UK, 1999. Springer-Verlag.

[17] Cormac Flanagan and Shaz Qadeer. A Type and Effect System for Atomicity. *SIGPLAN Not.*, 38(5):338–349, 2003.

[18] Dan Grossman. Type-safe Multithreading in Cyclone. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international works hop on Types in languages design and implementation*, pages 13–25, New York, NY, USA, 2003. ACM Press.

[19] Louis Mandel and Marc Pouzet. ReactiveML, a Reactive Extension to ML. In *ACM International conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July 2005.

[20] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly, 1996.

[21] Peter W. O'Hearn. Resources, Concurrency, and Local Reasoning (Abstract). In *ESOP*, pages 1–2, 2004.

[22] Marc Pouzet. Lucid Synchrone, version 3. Tutorial and Reference Manual. *Université Paris-Sud, LRI*, April 2006.

[23] John C. Reynolds. Separation Logic: a Logic for Shared Mutable Data Structures, 2002.

[24] Manuel Serrano, Frédéric Boussinot, and Bernard Serpette. Scheme Fair Threads. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 203–214, New York, NY, USA, 2004. ACM Press.

[25] Olivier Tardieu and Stephen A. Edwards. Scheduling-independent threads and exceptions in SHIM. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 142–151, New York, NY, USA, 2006. ACM Press.

[26] Jim Trevor, Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

Contents