

Mechanized Reasoning about Concurrent Functional Programs

Sava Mintchev

School of Computer Science and Information Systems Engineering,
University of Westminster,
115 New Cavendish Street,
London W1M 8JS,
UK

Tel: +44 171 911 5000

Fax: +44 171 911 5089

`S.M.Mintchev@westminster.ac.uk`

TR-CSPE-05, January 30, 1997

Abstract. Various mechanisms have been used for adapting functional languages to parallel machines, ranging from semantics-preserving annotations to concurrent language extensions. Concurrent extensions are applicable to both lazy and strict languages, and give the programmer full control over parallel evaluation; however, they complicate the proofs of program correctness.

This paper pursues the concurrent extension approach to parallel functional programming, and addresses the question of proving parallel programs correct with respect to sequential specifications. The paper presents an extension of a lazy functional language with concurrency primitives, allowing the dynamic creation of processes and point-to-point interprocess communication. The primitives are given an operational semantics, and an observational equivalence between processes is defined. The equivalence has been implemented in a theorem prover for concurrent functional programs. As an illustration, the derivation of a parallel program from a functional specification is given, and is proved correct with the theorem prover.

Keywords: functional programming, concurrency, theorem proving, program correctness

1 Introduction

Of all the different approaches to parallel programming in pure functional languages, *skeletons* and *concurrent extensions* seem to be the most practical. Algorithmic skeletons [2, 3] represent a structured approach to parallel programming, whereby parallelism is contained in a set of predefined higher-order functions. A major limitation is that the set of skeletons is fixed, and it may not always be appropriate for the problem at hand. Other approaches to parallel programming with high-order functional primitives can be considered as related to the skeleton approach, *e.g.* the use of the Bird–Meertens formalism for parallel programming [15].

Functional languages like Lisp and ML have been extended with concurrency and communication primitives. The resulting concurrent languages (*e.g.* Concurrent ML [14]) give the programmer full control over the parallel evaluation of programs; however, they complicate the task of proving a parallel program correct with respect to the sequential version.

This paper explores the concurrent extension approach to parallel functional programming, and its amalgamation with the skeleton approach. In particular, it addresses the problem of defining *new* skeletons and proving them correct. The proofs of correctness have been mechanized by an extended version of an induction theorem prover for lazy functional programs [10, 9].

We start with a small lazy functional language (the *Core* subset of Haskell), and extend it with concurrency primitives akin to those present in process calculi. The primitives and their semantics are introduced in Section 2. An observational equivalence between processes is defined in Section 3, and the issue of implementing the equivalence in the theorem prover is discussed. Finally, the use of the theorem prover in deriving a correct parallel program is demonstrated by example in Section 4.

2 A concurrent extension

The set of concurrency and communication primitives has been chosen with two objectives in mind: (i) easy reasoning about concurrent programs; (ii) easy implementation in a compiler/run-time system. All primitives appear in some form in process calculi like CSP [4], CCS [7], and Pi-calculus [8].

2.1 Primitives

Most modern process calculi use *channels* for communication. An alternative approach, found for example in early work on CSP, relies on point-to-point communication, with process identifiers (pids) used for addressing messages between processes. This latter approach has also been used in recent work on correctness of the parallel compilation of a functional language [17].

Most channel-based concurrent languages do not allow channel names to be passed on as values. A notable exception is the Pi-calculus, to which the idea of passing channel names between processes is central. It gives the Pi-calculus great flexibility in expressing systems with evolving structure, but at the price of a complicated execution mechanism (if the calculus were to be used as the basis for a programming language). In order to avoid the complications of channel passing, we have chosen to use process identifiers instead of channels. In this setup, every message is always sent to a single known receiver.

The typed functional language we take as a basis, *Core*, is a subset of Haskell having lambda abstractions and applications, *let* expressions, data constructors and *case* expressions (for taking data structures apart). The operational call-by-need semantics is given by a partial function, *red*, which reduces expressions to their

Result type	Primitive	Explanation
Process (<i>proc</i>)	$io @ pid$	process with identifier pid
	<i>act</i>	action
	<code>compose <i>procs</i></code>	parallel composition
	<code>result ($\lambda pid \rightarrow io$)</code>	make io a process
Input/output command (<i>io</i>)	<code>send $pid_{dest} msg io_{cont}$</code>	send to pid_{dest}
	<code>receive $pid_{src} (\lambda msg \rightarrow io_{cont})$</code>	receive from pid_{src}
	<code>spawn ($\lambda pid_{new} pid_{old} \rightarrow io_{new}$)</code> <code>($\lambda pid_{new} \rightarrow io_{old}$)</code>	spawn new process
	<code>die</code>	terminate
	<code>return e</code>	return result
	<code>$io_1 \oplus io_2$</code>	sum
Action (<i>act</i>)	<code>Send $pid_{dest} msg proc pid$</code>	
	<code>Receive $pid_{src} (\lambda msg \rightarrow proc) pid$</code>	
	<code>Die</code>	
	<code>Return e</code>	

Fig. 1. Concurrency and communication primitives

weak head normal forms. We shall also take for granted a semantic congruence relation (\equiv) between expressions in the functional language.

The language is enriched with three new primitive types: processes, actions (considered as a subtype of the type of processes), and input/output commands. Expressions of those types can be constructed using the concurrency primitives from Figure 1.

A *process* can have one of several forms. In its most basic form, a process consists of an input/output command (*io*) coupled with a process identifier (*pid*).

Input/output commands specify the communication action that a process must undertake. The `send` command sends a message msg to a process pid_{dest} , and continues with the command io_{cont} . The `receive` command receives a message from processor pid_{src} and passes it on to the continuation io_{cont} . The message can be any expression other than a process or action. The `spawn` command creates a new process with a fresh identifier pid_{new} , and invokes its code io_{new} giving it two process identifiers: its own and that of the parent process pid_{old} . The parent process is given the child's identifier, and continues with the execution of io_{old} . The `die` command causes a process to terminate. The `return` command is similar, but causes the process to return a result before terminating.

Two commands can be combined with a *sum* operator (\oplus), allowing the environment of a process to choose one of the two commands. If the environment allows either command to be executed, the choice is made nondeterministically.

The execution of an input/output command may result in a visible *action*. An action is a special kind of process having observable behaviour. The actions `Send`, `Receive`, `Die` and `Return` start with an uppercase letter so that they can be distinguished from the corresponding commands which cause them.

Processes can be combined in various way to form more complex processes with the help of combinators. The `compose` combinator forms the parallel composition of a list of processes. Finally, the `result` primitive can be used for converting an input/output command into a process by creating a new process identifier.

2.2 Commitment rules

A process may be able to commit various actions. The *commitment* (or transition) relation ($\succ \subseteq proc \times act$) associates processes with their possible actions. The

commitment rules are given in Figure 2. The rules contain an action that was not

Sending a message	
$(\text{send } pid_2 \text{ msg}_1 \text{ io}_1 @ pid_1) \succ (\text{Send } pid_2 \text{ msg}_1 (\text{io}_1 @ pid_1) pid_1)$	
$\frac{proc \succ (\text{Send } pid_2 \text{ msg}_1 proc_1 pid_1), pid_2 \notin procs}{\text{compose } (proc : procs) \succ (\text{Send } pid_2 \text{ msg}_1 (\text{compose } (proc_1 : procs)) pid_1)}$	
Receiving a message	
$(\text{receive } pid_1 (\lambda v \rightarrow io_2) @ pid_2) \succ (\text{Receive } pid_1 (\lambda v \rightarrow io_2 @ pid_2) pid_2)$	
$\frac{proc \succ (\text{Receive } pid_1 (\lambda v \rightarrow proc_{cont}) pid_2), pid_1 \notin procs, x \notin free(procs)}{\text{compose } (proc : procs) \succ (\text{Receive } pid_1 (\lambda x \rightarrow \text{compose}((proc_{cont}[x/v]) : procs)) pid_2)}$	
Communication	
$\frac{proc_1 \succ (\text{Send } pid_2 \text{ msg}_1 proc_{cont1} pid_1), proc_2 \succ (\text{Receive } pid_1 (\lambda v \rightarrow proc_{cont2}) pid_2)}{\text{compose } (proc_1 : proc_2 : procs) \succ \text{Tau } (\text{compose } (proc_{cont1} : (proc_{cont2}[msg_1/v]) : procs))}$	
$\frac{proc \succ \text{Tau } proc_{cont}}{\text{compose } (proc : procs) \succ \text{Tau } (\text{compose } (proc_{cont} : procs))}$	
Spawning a process	
$(\text{spawn } (\lambda v x \rightarrow io_{new}) (\lambda v \rightarrow io_{cont}) @ pid) \succ$ $\text{Tau } (\text{compose } [(io_{new}[pid_{new}/v, pid/x] @ pid_{new}), (io_{cont}[pid_{new}/v] @ pid)])$	
Termination of processes	
$(\text{die } @ pid) \succ \text{Die}; \quad \frac{\text{compose } procs \succ act}{\text{compose } (\text{Die} : procs) \succ act}; \quad \frac{(\text{return } e @ pid) \succ \text{Return } e}{\text{compose } [\text{Return } e] \succ \text{Return } e}$ $\text{compose } [\perp @ pid] \succ \text{Return } \perp$	
$\text{result } (\lambda pid \rightarrow io) \succ io[pid_{new}/pid] @ pid_{new}$	
Sum	
$\frac{(io_1 @ pid) \succ act}{(io_1 \oplus io_2 @ pid) \succ act}, \quad \frac{(io_2 @ pid) \succ act}{(io_1 \oplus io_2 @ pid) \succ act}$	
Parallel composition	
$\frac{\text{compose } (procs_1 \# procs_2) \succ act}{\text{compose } (\text{compose } procs_1 : procs_2) \succ act}, \quad \frac{\text{compose } (proc : (procs_1 \# procs_2)) \succ act}{\text{compose } (procs_1 \# [proc] \# procs_2) \succ act}$	
Actions	Reduction of Core expressions
$act \succ act$	$\frac{\text{red } io @ pid \succ act}{io @ pid \succ act}$

Fig. 2. Commitment rules

mentioned in Figure 1 — the *silent action* ($\text{Tau } proc$). This action accompanies any internal communication that takes place in a process.

The first set of commitment rules deals with the interpretation of send commands. A process commits a Send action upon execution of a send command. A parallel composition of a Send action with other processes $procs$ can commit that Send action (i.e. the Send can take place before any other actions of $procs$), provided that the destination process is not among $procs$. Receive actions are dealt

with in a similar way.

When a parallel composition of processes includes a pair of Send and Receive actions, internal communication can take place. In order for that to happen, the Send must have the Receive as its destination process, and the Receive must have the Send as its source. Notice that the internal communication causes a silent action that can be recorded by an external observer.

It is obvious from the commitment rules that a (Send pid_2 msg_1 $proc_{cont1}$ pid_1) process will block until the recipient is ready to receive. A non-blocking send can easily be modelled by spawning a new process for $proc_{cont1}$ before sending msg_1 .

The spawn command causes a new process to be created, with a fresh process identifier (pid_{new}), distinct from all other existing identifiers. Again, a silent action accompanies the creation of the process.

3 Observational equivalence

Now we need to define an observational equivalence relation on processes, in the style of [7]. That relation will provide a basis for the equivalence test implemented in the theorem prover.

3.1 Simulation and bisimilarity

First we introduce a modified commitment relation (\succ^*) that ignores silent (Tau) actions:

$$\frac{proc \succ act}{proc \succ^* act} (\forall proc_x. act \neq \text{Tau } proc_x); \quad \frac{proc_1 \succ \text{Tau } proc_2, proc_2 \succ^* act}{proc_1 \succ^* act}$$

We say that a process $proc_1$ is *simulated* by a process $proc_2$ if at any moment $proc_2$ can commit every action that $proc_1$ can. The simulating process $proc_2$ is allowed to ignore the silent actions of $proc_1$, or have additional silent actions. The notion of simulation is formalized in the following definition:

Definition 1. A binary relation \mathcal{S} is a *simulation* iff

For two ground processes $proc_1$ and $proc_2$, ($proc_1 \mathcal{S} proc_2$) implies that:

1. if $proc_1 \succ \text{Send } pid_{dest} \ msg \ proc_{cont1} \ pid$
then $\exists proc_{cont2}. (proc_2 \succ^* \text{Send } pid_{dest} \ msg \ proc_{cont2} \ pid) \wedge (proc_{cont1} \mathcal{S} proc_{cont2})$
2. if $proc_1 \succ \text{Receive } pid_{src} (\lambda v \rightarrow proc_{cont1}) \ pid$
then $\exists proc_{cont2}. (proc_2 \succ^* \text{Receive } pid_{src} (\lambda v \rightarrow proc_{cont2}) \ pid) \wedge (\forall e. proc_{cont1}[e/v] \mathcal{S} proc_{cont2}[e/v])$
3. if $proc_1 \succ \text{Die}$ then $proc_2 \succ^* \text{Die}$
4. if $proc_1 \succ \text{Return } e_1$ then $\exists e_2. (proc_2 \succ^* \text{Return } e_2) \wedge (e_1 \equiv e_2)$
5. if $proc_1 \succ \text{Tau } proc_{cont1}$ then $proc_{cont1} \mathcal{S} proc_2$

A binary relation \mathcal{S} is a *bisimulation* iff both \mathcal{S} and its opposite are simulations.

Bisimilarity (\approx) is defined as the largest bisimulation. Two ground processes are bisimilar if they are related by some bisimulation relation. Processes with free names are required to be bisimilar for all instantiations of the free names.

Bisimilarity as defined here corresponds to the notion of weak bisimilarity in CCS, since internal communications (silent actions) are disregarded when comparing two processes. Yet there is a difference — Definition 1 requires a simulating

process to terminate (die or return a result) whenever the simulated process terminates, while there is no such requirement in CCS. The difference can be illustrated by a process containing a cycle of internal communication. Consider the following input/output command definitions:

$$\begin{aligned} \mathit{init} &= \lambda p1 \rightarrow \mathit{spawn} \mathit{receiver} (\lambda p2 \rightarrow \mathit{sender} p1 p2) \\ \mathit{sender} &= \lambda p1 p2 \rightarrow \mathit{send} p2 0 (\mathit{sender} p1 p2) \\ \mathit{receiver} &= \lambda p2 p1 \rightarrow \mathit{receive} p1 (\lambda x \rightarrow \mathit{receiver} p2 p1) \end{aligned}$$

Any process which executes init will be locked in an infinite chain of communication between sender and $\mathit{receiver}$. Now consider the processes

$$\mathit{compose} [\mathit{init} p1 @ p1, \mathit{return} 0 @ p2] \quad \mathit{and} \quad (\mathit{return} 0 @ p2)$$

According to the definition of bisimilarity and the rules from Figure 2, these processes are not bisimilar, while the corresponding CCS agents would be weakly bisimilar. On the other hand, finite sequences of silent actions do not affect bisimilarity; for example, the pair of processes above would be bisimilar if the definitions of sender and $\mathit{receiver}$ were replaced by:

$$\begin{aligned} \mathit{sender} &= \lambda p1 p2 \rightarrow \mathit{send} p2 0 \mathit{die} \\ \mathit{receiver} &= \lambda p2 p1 \rightarrow \mathit{receive} p1 (\lambda x \rightarrow \mathit{die}) \end{aligned}$$

Bisimilarity is an equivalence relation, though not a congruence in general — the congruence fails for processes that are not *determinate*.

3.2 Determinacy

The concurrent processes we are particularly interested in have the same meaning as deterministic functional programs. The following definition gives a necessary condition for a process to be deterministic:

Definition 2. A process proc is (*locally*) *determinate* iff $\forall \mathit{proc}_1$,

$$\text{if } (\mathit{proc} \succ \mathit{tau} \mathit{proc}_1) \text{ then } \mathit{proc} \approx \mathit{proc}_1$$

Informally, a process is determinate if it cannot commit an internal action, or if all its possible internal actions leave it unchanged (up to bisimilarity). Note that determinacy as defined here is a local property, and is weaker than the notion in [7], where determinacy is a property of all descendants of an agent.

As an example of a determinate process, consider a process which executes the input/output command procP :

$$\begin{aligned} \mathit{procP} &= \lambda x y p1 \rightarrow \mathit{spawn} \mathit{procQ} (\lambda p2 \rightarrow \mathit{spawn} \mathit{procQ} (\lambda p3 \rightarrow \\ &\quad \mathit{send} p2 x (\mathit{send} p3 y \\ &\quad (\mathit{receive} p2 (\lambda x \rightarrow \mathit{receive} p3 (\lambda y \rightarrow \mathit{return} (x + y)))) \oplus \\ &\quad \mathit{receive} p3 (\lambda x \rightarrow \mathit{receive} p2 (\lambda y \rightarrow \mathit{return} (x + y)))))) \\ \mathit{procQ} &= \lambda p2 p1 \rightarrow \mathit{receive} p1 (\lambda x \rightarrow \mathit{send} p1 x \mathit{die}) \end{aligned}$$

The process spawns two new processes running procQ , and sends and receives from each one a value. The order in which it receives values back is unimportant because of the commutativity of arithmetic addition.

It is easy to see that determinacy is not preserved by \oplus : the sum of two determinate processes may not be determinate.

3.3 Expansion

The commitment rules from Figure 2 form the basis of a tactic for proving the equivalence of processes. The tactic is employed for extending the theorem prover from [10]. The main element of the tactic is the *expansion* of processes. To expand a parallel composition of processes means to replace it with the sum of all possible commitments of those processes.

Before we define expansion, we introduce a new *nondeterministic choice* operator, (\sqcap) , analogous to the one in CSP. It is described by the axiom:

$$(proc_1 \sqcap proc_2 \equiv proc_1) \vee (proc_1 \sqcap proc_2 \equiv proc_2)$$

The nondeterministic choice operator allows nondeterminism to be disentangled from communication. With (\sqcap) , nondeterminism caused by communication can be expressed without keeping track of silent (internal) actions.

Expanding a process which is not a parallel composition leaves that process unchanged. The expansion of a composition of processes is defined as:

$$\begin{aligned} \text{expand } (\text{compose } [proc_1, \dots, proc_n]) \\ & | \text{null } sends \wedge \text{null } recvs \wedge \text{null } comms \wedge \text{null } spawns \\ & \quad \vee (\text{irreducible process present}) = \text{compose } [proc_1, \dots, proc_n] \\ & | \text{otherwise} = \bigoplus (sends \cup recvs) \sqcap \prod (comms \cup spawns) \end{aligned}$$

where

$$\begin{aligned} sends &= \{ \text{Send } pid_{dest} \ msg_i \ (\text{compose } [proc_1, \dots, proc_{cont^i}, \dots, proc_n]) \ pid_i \\ & \quad | 1 \leq i \leq n, \\ & \quad \quad proc_i \succ \text{Send } pid_{dest} \ msg_i \ proc_{cont^i} \ pid_i, \ pid_{dest} \notin procs \} \\ recvs &= \{ \text{Receive } pid_{src} \\ & \quad (\lambda x \rightarrow \text{compose } [proc_1, \dots, proc_{cont^i}[x/v], \dots, proc_n]) \ pid_i \\ & \quad | 1 \leq i \leq n, \\ & \quad \quad proc_i \succ \text{Receive } pid_{src} \ (\lambda v \rightarrow proc_{cont^i}) \ pid_i, \ pid_{src} \notin procs \} \\ comms &= \{ \text{compose } [proc_1, \dots, proc_{cont^i}, \dots, proc_{cont^j}[msg_i/v], \dots, proc_n] \\ & \quad | 1 \leq i \leq n, 1 \leq j \leq n, \\ & \quad \quad proc_i \succ \text{Send } pid_j \ msg_i \ proc_{cont^i} \ pid_i, \\ & \quad \quad \quad proc_j \succ \text{Receive } pid_i \ (\lambda v \rightarrow proc_{cont^j}) \ pid_j \} \\ spawns &= \{ \text{compose } [proc_1, \dots, (io_{new}[pid_{new}/v, pid/x] @ pid_{new}), \\ & \quad \quad \quad (io_{cont}[pid_{new}/v] @ pid), \dots, proc_n] \\ & \quad | 1 \leq i \leq n, proc_i \succ (\text{spawn } (\lambda v \ x \rightarrow io_{new}) \ (\lambda v \rightarrow io_{cont}) @ pid) \} \\ procs &= [proc_1, \dots, proc_n] \end{aligned}$$

In an expanded process, internal communication is not accompanied by a silent action. Instead, all communications or spawns that can occur internally are gathered in a nondeterministic choice (\prod) , reflecting the fact that the external environment has no control over them. All Send and Receive actions directed at external processes are gathered in a sum (\bigoplus) , and the environment is free to choose one among them; if the set $(sends \cup recvs)$ is empty, the sum is omitted.

With this definition, expansion can actually *simplify* proofs about processes, because to prove $P(\prod proc_i)$, we actually prove $P(proc_i)$ for every i . The theorem prover uses expansion repeatedly in order to simplify parallel compositions of processes. To justify the use of expansion, we must show that it preserves bisimilarity of processes. In fact, it doesn't in general; expansion preserves bisimilarity only on determinate processes.

Proposition 3. *Expansion law:* a process $proc$ is determinate iff

$$\text{expand } proc \approx proc$$

To summarize: we can safely use expansion on a determinate process. If a process is nondeterminate, then expanding it results in a process which cannot be proved bisimilar to any process. Thus when the theorem prover uses expansion in testing two processes for bisimilarity, it may fail unless both processes are determinate.

4 An example: parallelizing the heat equation

In this section we demonstrate how a parallel version of a program can be proved correct with respect to a sequential specification. The proof has been carried out with a theorem prover for Core Haskell, extended with the concurrency primitives from the previous sections.

4.1 The sequential program

The example is a program to calculate the solution of the one-dimensional heat equation [11]. Although this is a concrete problem, the solution is applicable to a wide class of equations due to the use of higher-order functions.

In [11] an efficient program is derived from an inefficient ‘mathematical-style’ specification. That derivation has been repeated using the mechanical theorem prover, but here we start with a specification close to the final program from [11]:

$$solve = \lambda g \, vs \, xs \, ws \rightarrow aloopF (mscanNN \, g) [] \, vs \, xs \, ws$$

The function *solve* takes a function *g*, a vector of initial values *xs*, and two streams of boundary values (*vs*, *ws*). It produces a stream of vectors; each vector contains the temperatures of spatial points at a given moment in time. The argument function *g* calculates the current value of a point in space, given a triple of the previous values of that point and its two neighbours. The function *aloopF* (accumulating loop) iterates in time; *mscanNN* (nearest-neighbour scan) calculates the values of all points at a given moment in time.

$$\begin{aligned} aloopF \, f \, acc \, [] \, xs \, ws &= acc \, ++ \, [xs] \\ aloopF \, f \, acc \, vs \, xs \, [] &= acc \, ++ \, [xs] \\ aloopF \, f \, acc \, (v : vs) \, xs \, (w : ws) &= \\ & aloopF \, f \, (acc \, ++ \, [xs]) \, vs \, (f \, (T \, v \, xs \, w)) \, ws \end{aligned}$$

$$\begin{aligned} mscanNN &= \lambda f \, (T \, v \, xs \, w) \rightarrow \\ & \text{case } xs \text{ of} \\ & [] \rightarrow [] \\ & x : xs \rightarrow f \, (T \, v \, x \, (shiftLout \, xs \, w)) : mscanNN \, f \, (T \, x \, xs \, w) \end{aligned}$$

Here the constructors *P* and *T* are used to build pairs and triples, respectively. The function *mscanNN* uses vector shifting functions, defined as:

$$\begin{aligned} shiftRout &= \lambda v \, xs \rightarrow sndP \, (shiftR \, v \, xs) \\ shiftLout &= \lambda xs \, w \rightarrow fstP \, (shiftL \, xs \, w) \\ shiftR &= \lambda x \, xs \rightarrow \text{let } i = \text{length } xs \text{ in} \\ & P \, (\text{take } i \, (x : xs)) \, (\text{head } (\text{drop } i \, (x : xs))) \\ shiftL &= \lambda xs \, x \rightarrow P \, (\text{head } (xs \, ++ \, [x])) \, (\text{tail } (xs \, ++ \, [x])) \end{aligned}$$

4.2 The parallel program

Now we set out to parallelize this program. The basic idea is to partition the vector of spatial points into a number of component vectors, and to place each component vector on a separate processor. A processor will then be responsible for repeatedly calculating its vector at successive moments in time. In order to do that, each processor will have to communicate with its neighbours (the processors holding neighbouring component vectors) at every time step. In fact we only need to modify the function *aloopF*; the other functions remain unchanged.

The parallel version of *aloopF* is illustrated in Figure 3. There are two terminal

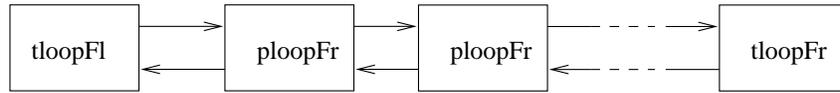


Fig. 3. Parallel loop

(boundary) communicating processes, *tloopFl* and *tloopFr*, and an arbitrary number of intermediate processes, *ploopFr*. The definitions of these processes are given below:

```

tloopFl = λ f acc vs xs pid rpid →
  case vs of
  [] → receive rpid (λ ys → return (zipWith (+) (acc ++ [xs]) ys))
  v : vs → send rpid (shiftRout v xs) (receive rpid (λ w →
    tloopFl f (acc ++ [xs]) vs (f (T v xs w)) pid rpid))
  
```

```

ploopFr = λ f acc n xs pid lpid rpid →
  case n of
  0 → receive rpid (λ ys → send lpid (zipWith (+) (acc ++ [xs]) ys) die)
  Succ n → receive lpid (λ v → send lpid (shiftLout xs err)
    (send rpid (shiftRout err xs) (receive rpid (λ w →
      ploopFr f (acc ++ [xs]) n (f (T v xs w)) pid lpid rpid))))
  
```

```

tloopFr = λ f acc xs ws pid lpid →
  case ws of
  [] → send lpid (acc ++ [xs]) die
  w : ws → receive lpid (λ v → send lpid (shiftLout xs w)
    (tloopFr f (acc ++ [xs]) (f (T v xs w)) ws pid lpid))
  
```

The left and right boundary processes, *tloopFl* and *tloopFr*, take as arguments the left and right boundary streams (*vs* and *ws*) respectively. Each process is given its own process identifier, *pid*, as an argument. An intermediate process *ploopFr* also needs the identifiers of its left (*lpid*) and right (*rpid*) neighbours, so that it can communicate with them, and the number of iterations *n* which is expected to be equal to the lengths of the boundary streams. The *(zipWith(+))* function takes two lists of lists and concatenates corresponding list elements to produce a single ‘zipped’ output list.

A separate function, *spawnLoops*, does the initialization by spawning off as many copies of *ploopFr* as desired.

```

spawnLoops = λ f xs ws pid lpid →
  case xs of
  [] → tloopFr f [] xs ws pid lpid
  x : xs → let p = split xs in
            spawn (spawnLoops f (sndP p) ws)
                  (ploopFr f [] (length ws) (x : fstP p) pid lpid)

```

The definition of *spawnLoops* uses the function *split* which splits up a list into two parts. We do not give an implementation of *split* here, but require that any implementation satisfy:

$$\forall bs. (fstP (split bs) ++ sndP (split bs)) \equiv bs$$

Now the parallel version of *solve* is:

```

psolve = λ g vs xs ws pid → let p = split xs in
  spawn (spawnLoops (mscanNN g) (sndP p) ws)
        (tloopFl (mscanNN g) [] vs (fstP p) pid)

```

4.3 Correctness proof

Now we want to prove that the parallel composition of *tloopFl*, *ploopFr* and *tloopFr* processes calculates the same result as the original sequential function *aloopF*. We do this in several stages. First of all we show that the composition of single instances of *ploopFr* and *tloopFr* can be replaced by a single instance of *tloopFr*, as shown in Figure 4¹.



Fig. 4. Composing *ploopFr* and *tloopFr*

Theorem 4.

$$\begin{aligned}
& \forall f. [decomposable\ f, \ nonnull\ f] \Rightarrow \\
& (\forall ws. \forall vs. \forall ass. \forall bss. \forall as. \\
& [length\ vs \equiv length\ ws, \ null\ as \equiv False, \ length\ bss \equiv length\ ass] \Rightarrow \\
& (\forall bs. \forall lpid. \forall p1. \forall p2. \\
& \quad compose\ [ploopFr\ f\ ass\ (length\ vs)\ as\ p1\ lpid\ p2\ @\ p1, \\
& \quad \quad tloopFr\ f\ bss\ bs\ ws\ p2\ p1\ @\ p2]) \\
& \equiv \\
& \quad compose\ [tloopFr\ f\ (zipWith\ (+)\ ass\ bss)\ (as\ ++\ bs)\ ws\ p1\ lpid\ @\ p1])
\end{aligned}$$

where

$$\begin{aligned}
decomposable\ f &= (\forall s. \forall t. \forall xs. \forall ys. f\ (T\ s\ (xs\ ++\ ys)\ t) \equiv \\
& \quad (f\ (T\ s\ xs\ (shiftLout\ ys\ t))\ ++\ f\ (T\ (shiftRout\ s\ xs)\ ys\ t))) \\
nonnull\ f &= (\forall zs. [null\ zs \equiv False] \Rightarrow \\
& \quad (\forall p. \forall r. null\ (f\ (T\ p\ zs\ r)) \equiv False))
\end{aligned}$$

¹ Note that the symbol (\equiv) is overloaded: it stands for (\approx) in the context of processes.

As can be seen from the premises of Theorem 4, the function f must possess certain properties. In the first place, an application of f should be decomposable by splitting up the list argument into two lists xs and ys , and appending the results of applying f to xs and ys . Secondly, when given a nonempty list argument zs , the function f should return a nonempty result. The theorem is proved by induction² on the boundary stream ws .

By repeated application of Theorem 4 we can reduce a parallel composition of any finite number of $ploopFr$ processes with $tloopFr$ into a single $tloopFr$ process. In other words, we can show that the function $spawnLoops$, which starts up a number of parallel $ploopFr$ processes, is equivalent to a $tloopFr$. This fact is stated in the following theorem:

Theorem 5.

$$\begin{aligned} & \forall f. [\text{decomposable } f, \text{nonnull } f] \Rightarrow \\ & (\underline{\forall} n. \forall xs. [(length\ xs < n) \equiv True] \Rightarrow \\ & \quad (\forall lpid. \forall pid. \forall ws. \\ & \quad \quad \text{compose } [spawnLoops\ f\ xs\ ws\ pid\ lpid\ @\ pid] \\ & \quad \quad \equiv \\ & \quad \quad \text{compose } [tloopFr\ f\ []\ xs\ ws\ pid\ lpid\ @\ pid])) \end{aligned}$$

The proof amounts to a well-founded induction on the length of the vector xs . Next we have to show that the parallel composition of $tloopFl$ and $tloopFr$ is equivalent to the original sequential function, $alooF$:

Theorem 6.

$$\begin{aligned} & \forall f. [\text{decomposable } f] \Rightarrow \\ & (\underline{\forall} ws. \forall vs. \forall ass. \forall bss. \\ & \quad [length\ vs \equiv length\ ws, length\ bss \equiv length\ ass] \Rightarrow \\ & \quad (\forall as. \forall bs. \forall p1. \forall p2. \\ & \quad \quad \text{compose } [tloopFl\ f\ ass\ vs\ as\ p1\ p2\ @\ p1, \\ & \quad \quad \quad tloopFr\ f\ bss\ bs\ ws\ p2\ p1\ @\ p2] \\ & \quad \quad \equiv \\ & \quad \quad \text{compose}[\text{return}(alooF\ f(\text{zipWith } (+)\ ass\ bss)\ vs\ (as++bs)\ ws)\ @\ p1])) \end{aligned}$$

We have now established the correctness of the parallel loop functions for any *decomposable* and *nonnull* function f . The last remaining proof obligation is to show that (*decomposable* ($mscanNN\ g$)) and (*nonnull* ($mscanNN\ g$)) hold for any argument function g . Then the main theorem follows directly.

Theorem 7. Correctness of psolve

$$\begin{aligned} & \forall vs. \forall ws. \forall xs. [length\ vs \equiv length\ ws, null\ xs \equiv False] \Rightarrow \\ & \quad (\forall g. \text{result } (psolve\ g\ vs\ xs\ ws) \equiv \text{Return } (solve\ g\ vs\ xs\ ws)) \end{aligned}$$

Now we can be sure that the parallel version of our program will produce the same results as the initial sequential version. What about the efficiency? Looking at the definitions of the parallel loop functions and at Figure 3, we can see that the parallel program will not be very fast. Since every communication requires a handshake, a long chain of $ploopFr$ processes will behave much like a ripple-carry adder: the leftmost $ploopFr$ reads a value from $tloopFl$, then passes a value to its neighbour on the right, which in turn passes it on, etc; and the same effect is then observed in the opposite direction. One way of avoiding this ‘ripple-carry’ effect is to introduce a process, $ploopFl$, whose communication pattern is a ‘mirror image’ of $ploopFr$.

² The use of induction in this theorem and elsewhere is indicated by underlining the universal quantifier for the induction variable.

When *ploopFr* expects to receive a value from its left neighbour, *ploopFl* sends a value to its right neighbour and *vice versa*:

```

ploopFl = λ f acc n xs pid lpid rpid →
  case n of
  0 → receive rpid (λ ys → send lpid (zipWith (++) (acc++[xs]) ys) die)
  Succ n → send rpid (shiftRout err xs) (receive rpid (λ w →
    receive lpid (λ v → send lpid (shiftLout xs err)
      (ploopFl f (acc ++ [xs]) n (f (T v xs w)) pid lpid rpid))))

```

Replicas of *ploopFl* can be fitted in between every pair of *ploopFr* processes, as shown in Figure 5. We must show that the introduction of *ploopFl* does not

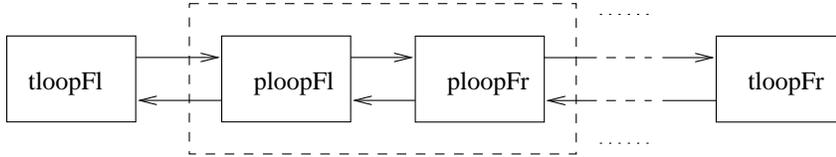


Fig. 5. Parallel loop

change the final result. More specifically, we prove that the parallel composition of *ploopFl* with *tloopFr* can be replaced by a single instance of *tloopFr*. The theorem is analogous to Theorem 4 and is omitted for brevity.

With this we conclude the heat equation example. We have proved the correctness of a parallelized version of a higher-order function (*alooF*). The function is an example of a ‘geometric paradigm’ skeleton, and thus we have demonstrated how a provably correct algorithmic skeleton can be constructed. We have also shown how the side conditions attached to the skeleton can be verified when the skeleton is instantiated, resulting in a correct parallel program.

5 Conclusion and related work

We presented a concurrent extension to a pure functional language, allowing programs to be composed of communicating processes. An equivalence relation on processes was defined, and an equivalence test was implemented in a theorem prover. Mechanized reasoning about parallel programs was demonstrated by example.

The concurrent extension approach to parallel functional programming is lower-level than the algorithmic skeleton approach. But of course a programmer is not prevented from defining skeletons in an explicitly concurrent functional language, and using just those skeletons in application programs. With a theorem prover, the user-defined skeletons can be verified; furthermore, any conditions for the applicability of particular skeletons can be validated wherever they are used.

A lot of work has been done on concurrent extensions to functional languages, but most of it is focused on implementations. Jones and Hudak describe in [5] explicitly concurrent programming in Haskell based on an extension to the Haskell I/O monad; however they do not give a formal semantics of the concurrency primitives. *Concurrent Haskell* [6] is a recently proposed extension, allowing concurrent programs to be written in a monadic style. Another non-strict language allowing communicating processes is Concurrent Clean [12].

Combinations of concurrent and strict functional languages include Concurrent ML [14] and Facile [13]. A formal operational semantics of a Concurrent ML-like language is given in [1, 14]. A sound type system for the concurrent language is presented in [14], and a type system for Facile is given in [16]. In the present paper typing issues have been ignored, and attention has been focused instead on the more general problem of reasoning about the behaviour of programs.

References

1. D. Berry, R. Milner, and D.N. Turner. A semantics for ML concurrency primitives. In *Principles of Programming Languages*, Jan 92.
2. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.
3. J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel programming using skeleton functions. In *Parallel Architectures and Languages Europe*. Springer-Verlag, June 93.
4. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science, 1985.
5. M.P. Jones and P. Hudak. Implicit and explicit parallel programming in Haskell. Technical Report YALEU/DCS/RR-982, Yale University, 1993.
6. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of the ACM Symposium on Principles of Programming Languages, St Petersburg Beach, Florida, January 1996*. ACM, 1996.
7. R. Milner. *Communication and Concurrency*. Prentice-Hall International Series in Computer Science, 1989.
8. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I and II. *Information and Computation*, 100(1):1–77, September 1992.
9. S. Mintchev. *Machine-Supported Reasoning about Functional Language Programs and Implementations*. PhD thesis, University of Manchester, Department of Computer Science, October 1995.
10. S. Mintchev. Mechanized reasoning about functional programs. In K. Hammond, D.N. Turner, and P. Sansom, editors, *Functional Programming, Glasgow 1994*, pages 151–167. Springer-Verlag Workshops in Computing, 1995.
11. J. O’Donnell and G. Runger. A case study in parallel program derivation: the heat equation algorithm. In K. Hammond, D.N. Turner, and P. Sansom, editors, *Functional Programming, Glasgow 1994*, pages 167–183. Springer-Verlag Workshops in Computing, 1994.
12. R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.
13. S. Prasad, A. Giacalone, and P. Mishra. Operational and algebraic semantics for Facile: a symmetric integration of concurrent and functional programming. In M.S. Paterson, editor, *17th International Colloquium on Automata, Languages and Programming (ICALP’90)*, LNCS 443, pages 765–781. Springer-Verlag, 1990.
14. J.H. Reppy. Concurrent ML: Design, application and semantics. In *Programming, Concurrency, Simulation and Automated Reasoning*. Springer-Verlag, LNCS 693, 1992.
15. D. B. Skillicorn. The Bird-Meertens formalism as a parallel model. In J.S. Kowalik and L. Grandinetti, editors, *NATO ARW “Software for Parallel Computation”*, volume 106 of *Series F*. NATO ASI Workshop on Software for Parallel Computation, Cetraro, Italy, June 1992, Springer-Verlag NATO ASI, 1993.
16. B. Thomsen. Polymorphic sorts and types for concurrent functional programs. In John Glauert, editor, *Proceedings of the 6th International Workshop on the Implementation of Functional Languages*, 7-9 September 1994, UEA Norwich, UK, 1994.
17. M. Wand. Compiler correctness for parallel languages. In *Functional Programming and Computer Architecture (FPCA)*. ACM, 1995.