Parallel Functional Programming for Computer Algebra

Dissertation

zur Erlangung des akademischen Grades
"Doktor der technischen Wissenschaften"

eingereicht von

Dipl.-Ing. Wolfgang Schreiner

1. Begutachter: o.Univ.Prof. Dr. Bruno Buchberger
2. Begutachter: o.Univ.Prof. Dr. Jens Volkert

Angefertigt am
Forschungsinstitut für Symbolisches Rechnen
Technisch-Naturwissenschaftliche Fakultät
Johannes Kepler Universität Linz

September 1994

Lebenslauf

Ich wurde am 28. September 1967 in Linz geboren und bin österreichischer Staatsbürger. Ich bin ledig und habe keine Kinder. Meine Eltern sind Anton Schreiner, Arbeiter, und Maria Schreiner, Hausfrau.

Von 1974 bis 1978 besuchte ich die Volksschule 28 in Linz, von 1978 bis 1986 das Bundesgymnasium Ramsauerstraße wo ich 1986 mit ausgezeichnetem Erfolg maturierte. Nach Abschluß des Gymnasiums begann ich das Studium der Informatik an der Johannes Kepler Universität Linz, das ich 1990 mit der mit Auszeichnung bestandenen Diplomprüfung und der Sponsion zum Diplom-Ingenieur beendete.

Meine Diplomarbeit verfaßte ich am Forschungsinstitut für Symbolisches Rechnen (RISC-Linz) unter der Betreuung von Prof. Dr. Buchberger. Für diese Diplomarbeit erhielt ich den OCG-Förderpreis 1991 der Österreichischen Computer Gesellschaft. Weiters wurde mir in Anerkennung hervorragender Studienleistungen der Würdigungspreis 1991 des Bundesministers für Wissenschaft und Forschung verliehen.

Seit 1991 bin ich Doktoratsstudent am RISC-Linz und Vertragsassistent im halben Beschäftigungsausmaß an der Universität Linz. Für die UNISOFTWARE PLUS GmbH habe ich 1992 als freier Mitarbeiter ein Seminar über Paralleles Rechnen erstellt. Weiters habe ich im Organisations-Komitees der CONPAR 94/VAPP VI International Conference on Parallel and Vector Processing mitgewirkt.

Eidesstattliche Erklärung

Ich versichere, daß ich die Dissertation selbständig verfaßt habe, andere als die angegebenen Quellen und Hilfsmittel nicht verwendet und mich auch sonst keiner unerlaubten Hilfe bedient habe.

Wolfgang Schreiner

Meinen Eltern

Abstract

The efficient implementation of parallel computer algebra algorithms puts heavy demands on the underlying language and runtime kernel: Because of the dynamic nature of the algorithms, the system must provide the programmer with very flexible forms of task creation, communication, and synchronization. Because of the dynamic nature of the basic objects, the system must support automatic heap management with garbage collection. On the other hand, the parallel programming model should be as high-level as possible to narrow the gap between the notation in which the mathematical theory is formulated and the language in which the algorithms are implemented.

This dissertation develops a software environment for parallel algebraic computation consisting of the PACLIB kernel for task and memory management and a compiler for the functional language pD where the programmer may specify parallelism in various levels of abstractions.

The PACLIB kernel is upward compatible to the computer algebra library SACLIB which can be immediately used in the parallel context. Several forms of task scheduling (eager and lazy task creation) and a flexible non-deterministic synchronization/communication mechanism are supported. In a case study, we formally verify this scheme using the temporal logic framework that we generalize for our purpose. This kernel provides an interface to the C language and thus implements in its own a programming model that has been already used for implementing a C library of parallel computer algebra algorithms.

We have developed a compiler for pD that generates efficient C code from functional specifications. A "pseudo-functional" extension is defined to access the parallel kernel features from the functional layer. We introduce a set of annotations by which the programmer may specify parallelism in the functional context. The compiler generates from these annotations explicit pseudo-functional task creation and synchronization operations; the C target code does not require further runtime checks. Furthermore, we define "parallel compositions" as a high-level notion to express typical forms of parallel algorithms. These compositions are compiled into parallel annotations or pseudo-functional operations; we analyze their dynamical behavior in detail.

The system has been implemented on a 20 processor Sequent Symmetry shared memory parallel computer. We have parallelized several non-trivial computer algebra algorithms in pD; the efficiency of the programs is very close to manually coded parallel C programs.

Keywords: parallel computation, functional programming, computer algebra, task management, garbage collection, temporal logic, compilation, shared memory multi-processors.

Zusammenfassung

Die effiziente Implementierung paralleleler Computeralgebra-Algorithmen stellt hohe Anforderungen an Programmiersprache und Laufzeitsystem. Wegen der dynamischen Algorithmen muß das System flexible Formen der Erzeugung, Kommunikation und Synchronisation paralleler Tasks unterstützen. Wegen der dynamischen Objekte muß das System einen Heap-Verwaltung mit automatischer Speicherbereinigung zur Verfüng stellen. Andererseits sollte das parallele Programmiermodell so abstrakt wie möglich sein, um die Sprache in der ein Algorithmus implementiert wird der Notation anzunähern in der die mathematische Theorie formuliert wird.

Diese Dissertation entwickelt eine Software-Umgebung für parallele Computeralgebra, bestehend aus dem PACLIB Kern für Task- und Speicherverwaltung und einem Übersetzer für die funktionale Programmiersprache pD, in der Parallelität auf verschiedenen Abstraktionsebenen beschreiben werden kann.

Der PACLIB Kern ist kompatibel mit der Computeralgebra-Bibliothek SACLIB, die damit unmittelbar im parallelen Umfeld eingesetzt werden kann. Verschiedene Formen von Task-Scheduling und ein nicht-deterministischer Synchronisations- und Kommunikations-Mechanismus werden unterstützt. In einer Fallstudie wird dieses Schema formal verifiziert, wobei wir den Rahmen der temporalen Logik für unsere Zwecke erweitern. Dieser Kern stellt eine Schnittstelle zur Programmiersprache C zur Verfügung und realisiert damit selbst ein Programmiermodell, das bereits für die Implementierung einer Bibliothek paralleler Computeralgebra-Algorithmen eingesetzt worden ist.

Wir haben einen Compiler für pD entwickelt, der effizienten C Code aus funktionalen Spezifikationen erzeugt. Eine "pseudo-funktionale" Erweiterung wird für den Zugriff auf den parallelen Kern definiert. Wir führen eine Anzahl von Annotationen ein, durch die der Programmierer Parallelität im funktionalen Umfeld ausdrücken kann. Der Compiler generiert aus diesen Annotationen explizite pseudo-funktionale Operationen für die Erzeugung und Synchronisation von Tasks; der erzeugte C Code erfordert keine weiteren Laufzeit-Prüfungen. Weiters definieren wir "parallele Zusammensetzungen" als eine abstraktere Möglichkeit, um typische Formen paralleler Algorithmen auszudrücken. Diese Formen werden in parallele Annotationen oder pseudo-funktionale Operationen übersetzt; wir analisieren ihr dynamisches Verhalten im Detail.

Das System wurde auf einem Sequent Symmetry Parallelrechner mit 20 Prozessoren und gemeinsamem Speicher implementiert. Wir haben verschiedene nicht triviale Computeralgebra-Algorithmen parallelisiert; die Effizienz der Programme entspricht der von manuell programmierten C Programmen.

Schlüsselwörter: paralleles Rechnen, funktionales Programmieren, Computeralgebra, Task-Verwaltung, Speicher-Verwaltung, temporale Logik, Übersetzung, Multi-Prozessoren mit gemeinsamem Speicher.

Contents

List of Figures

Acknowledgements

First of all, I'd like to thank Bruno Buchberger, my thesis supervisor and chair of the RISC-Linz institute. He has given me the possibility to work in a stimulating atmosphere on this interesting and fruitful research area. His personality as a researcher, organizer and a human have always been a model for me.

I am indebted to Hoon Hong for many discussions, technical and philosophical ones, which helped me to shape my understanding on language design and research in general and to focus the thesis on the demands of computer algebra. Hoon has triggered the development of the PACLIB kernel as an entity of its own with a ready to use programming interface.

I owe much to the RISC-Linz parallel computation group. Hans-Wolfgang Loidl (who spends now a research year in Glasgow) has joined my interest in parallel functional programming since the time of our diploma thesis work. He was also the first one to test the D language by implementing a linear equation solver. Kurt Siegl and his ||MAPLE|| system have contributed to a stimulating "competition" of different approaches to parallel computer algebra.

The STURM project group directed by Hoon Hong, in particular Andreas Neubacher, Volker Stahl, and Pier Giorgio Bertoli, have provided substantial input and feedback to the design of the PACLIB kernel and its imperative programming interface.

Last but not least, I'd like to express my warmest thanks to my parents for their support during the past years without which this thesis would not have been possible.

The Austrian Science Foundation (FWF) has supported my work for this dissertation by the grant S5302-PHY "Parallel Symbolic Computation".

Software Credits

Academic work today heavily relies on free software developed by idealistic computer scientists and engineers in their spare time, mostly not for money but the sake of the public. Since my daily work and this thesis heavily rely on their products, I wish to give them all the credit they deserve:

- **Free Software Foundation.** Richard Stallman and the FSF for all the software distributed under the GNU Copyleft, in particular the GNU Emacs editor, the GNU C compiler gcc, the GNU debugger gdb, the GNU parser generator bison, the GNU lexical analyzer generator flex.
- **TeX, LaTeX, BibTeX.** Donald Knuth for his type-setting system TeX and Leslie Lamport for his macro package LaTeX with which this document was prepared; Oren Patashnik for the bibliography package BibTeX.
- **Linux.** Linus Torvald for his operating system kernel Linux which allowed me to run all the free software available for Unix on my private PC; all others that helped to make Linux a complete operating system.
- **X11.** The (originally MIT) X Consortium for the X11 window system without which any Unix system would be worth only the half.
- **xxgdb.** Po Cheung and Pierre Willard for the xxgdb debugger interface which essentially helped me reduce debugging times.
- **xdvi.** Eric Cooper, Bob Scheifler and Mark Eichin for the previewer xdvi which I extensively used when preparing this document.
- **dvips.** Thomas Rokicki's and Don Knuth's DVI to PostScript converter dvips which allowed me to print this thesis.
- **xfig.** Supoj Sutanthavibul, Paul King and many others for the drawing program xfig with which most figures in this thesis were produced.
- **fig2dev.** Micah Beck, Frank Schmuck and Conrad Kwok for their package fig2dev for the translation of Fig figures into PostScript.
- **gnuplot.** Colin Kelley and Thomas William plotting program gnuplot with which the diagrams in Section 7.7 were produced.
- **Ghostscript and ghostview.** L. Peter Deutsch and Aladdin Enterprises for the PostScript interpreter Ghostscript and Erik M. van der Poel for the associated previewer ghostview with which I previewed the figures and diagrams in this thesis.
- **RCS.** Walter F. Tichy for the Revision Control System RCS that helped me to bring order into the chaos of software and document management.
- **ispell.** Pace Willisson, Wait Bühring, Geoff Künning and Bob Devine for the spelling checker ispell which helped me to correct (hopefully most) spelling errors in this thesis.
- **Others.** All others for the tools I use for my daily work.

Thank you all.

The PACLIB kernel described in Chapter 2 is an extension of the runtime system for the computer algebra library **SACLIB** [Buchberger *et al.*, 1992]; its task management is built on top of the **uSystem** package [Buhr and Stroobosscher, 1990]. The **Standard ML** language [Appel and MacQueen, 1991] is used in Section 4.6 as a comparison basis for benchmarks of several D programs.

Chapter 1

Introduction

Computer algebra is that area in mathematics respectively computer science that aims at exact solutions of problems in natural and technical sciences [Buchberger *et al.*, 1982]. Research results of this area are e.g. algorithms for symbolic integration, polynomial factorization or the exact solution of algebraic equations and inequalities. These algorithms operate with arbitrary precision arithmetic; they are therefore more expensive with respect to time and space than the corresponding numerical methods. However, they always provide correct solutions and are not influenced by the stability properties of the given problem.

While most computer algebra algorithms are based on the concept of mathematical functions, computer algebra programs that are intended for computationally intensive applications are usually written in a rather low-level imperative style (typically in C). Consequently, there is little resemblance between the notation in which the mathematical theory is formulated and the language in which the code is actually programmed.

In particular, it is difficult to perform the restructuring necessary to exhibit the parallelism contained in a problem i.e. to transform a sequential algorithm into a parallel one. Because of low-level programming details, it is a rather time-consuming process to experiment with different parallelization strategies or different variants of a parallel algorithm.

In this dissertation, we describe a programming style where the programmer is freed from some of these technical details and may more concentrate on the essential tasks. The part of the programmer is to formulate a parallel algorithm in a functional notation and mark some critical spots. The part of the compiler is to create the necessary task creation and synchronization statements such that the denoted parallelism is efficiently utilized.

We present the design and implementation of pD, a parallel variant of a functional language with strict execution semantics. In particular, we elaborate the compilation techniques that we apply to translate a pD program with para-functional annotations into statically typed parallel C code that corresponds (with restrictions) in structure and quality to a manually coded explicitly parallel program.

The target code is linked with the PACLIB kernel, a multi-processor variant of the runtime system for the computer algebra library SACLIB [Buchberger *et al.*, 1992]. This allows to call SACLIB functions implemented in C from pD programs or, vice versa, call pD functions from the C layer of SACLIB/PACLIB. Consequently (parallel) modules may be implemented on the imperative language layer and/or on the functional layer and freely combined in an application.

Besides embedability into an existing environment, two main goals with the development of pD were

1. **Efficiency.** pD is intended for parallel programming. The only motivation for this is to reduce the runtime compared to an efficient sequential program. Assuming a limited number of processors, we therefore cannot afford to lose sequential execution efficiency. The basis of our comparison is the speed of sequential SACLIB C programs, respectively of corresponding parallel PACLIB C variants.

2. **Flexibility.** pD is intended for computer algebra. This area is based on dynamic data structures and algorithms that demand very flexible forms of parallelism. Parallel programming based on pure data parallelism (the SIMD/SPMD model) is not sufficient; only those approaches are appropriate that allow very flexible forms of synchronization.

We chose the functional approach to high-level parallel programming because computer algebra is based on the concept of mathematical functions and most computer algebra programs are functional in spirit (even if implemented in C). However, we are not religious about this approach; e.g. pD also supports non-determinism, which may in some cases help to significantly reduce synchronization delays. On a higher level, this non-determinism is however embedded in a referential transparent framework.

The main *original features* of this approach are the following:

1. The integration of several abstractions levels for parallel programming all of which are open to the programmer. While most approaches in (para-)functional programming tend to stress the differences to (parallel) imperative programming, we attempt to emphasize the similarities showing how the abstractions of one level translate into the abstraction of the next level. The lower levels give more flexibility but the higher levels allow to hide details. It is the decision of the programmer which level is appropriate.

2. The compilation of para-functional annotations into a statically typed parallel program. On the user-level, the para-functional program is transparent with respect to tasks (i.e. function interfaces need not be changed and synchronization instructions need not be inserted). On the target level, however the program is statically typed (i.e. no runtime task checks are required). Still by aggressive deferring of synchronizations most of the parallelism in functional languages based on futures is preserved.

The first issue distinguishes pD from closed systems where the programmer is forced to operate on a single level of abstraction which the language designer considers as the only saving one. We hope that in general our abstractions are useful; however, if they are not appropriate, we at least give way to the ideas of the programmer.

The second issue distinguishes our approach from those systems that only support very restricted forms of parallelism. Since our intended area of application (symbolic algebra) is based on very dynamic algorithms and data structures, we also need very flexible forms of task creation and synchronization. It also distinguishes our approach from those systems that sacrifice efficiency by e.g. inserting at strict basis operations synchronization checks.

There are several related projects going on in different research communities (see Chapter 9 for a comprehensive overview):

Research groups on computer algebra have developed parallel variants of computer algebra packages, e.g. PAC [Roch, 1988], PARSAC [Küchlin, 1990a], Maple/Linda [Char, 1990] all of which are based on parallel variants of C. ||MAPLE|| (parallel Maple) [Siegl, 1993a] is based on an interpreter for the declarative (guarded Horn clause) language STRAND.

Research groups on functional programming have developed parallel variants of functional languages (for a bibliography, see e.g. [Schreiner, 1993b]). Unfortunately, there is still

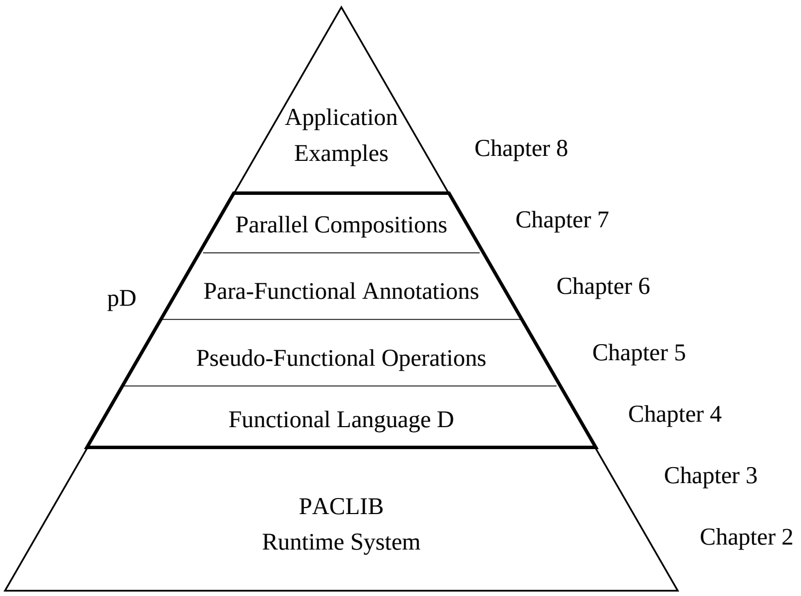


Figure 1.1: Thesis Outline

a strong tradeoff between efficiency and flexibility. Among the more efficient approaches are SISAL [Feo *et al.*, 1990], various data-parallel versions of functional languages [Blelloch, 1993], and several skeleton-based approaches [Cole, 1988]. These languages are often intended for numerical ("scientific") applications.

Among the approaches that put more emphasis on flexibility are various dialects of Lisp based on "futures" [Halstead, Jr., 1985], dataflow languages [Nikhil, 1991], and para-functional variants of (typically lazy) functional languages [Hudak, 1986b]. These languages support dynamic task creation but in general rely on hardware support or runtime checks for transparent task synchronization.

On the contrary, this dissertation presents a scheme where the user may program in a para-functional (or future-based) style and the compiler analyzes the program to automatically insert the synchronization operations. Synchronization is deferred as far as possible (even beyond function and task boundaries) to promote parallelism. The imperative target code is statically typed and runtime type checks can be avoided.

The rest of the thesis is organized as follows (see Figure 1.1):

Chapter 2 (The PACLIB Kernel) describes the runtime kernel of the system that provides several forms of task scheduling (eager and lazy task creation) and flexible non-deterministic synchronization.

Chapter 3 (The Correctness of the PACLIB Kernel) verifies the non-deterministic synchronization algorithm by a formal correctness proof. We introduce extensions to the standard temporal logic framework to make it more suitable for our purpose.

Chapter 4 (The Functional Language) describes D, the "sequential" subset of pD and its compilation into efficient C code; special optimizations are introduced for the compilation of list-generating functions into loops.

Chapter 5 (The Pseudo-Functional Language) describes the interface between the kernel and the functional layer by new types `Task` and `Stream` and operations for explicit task creation and synchronization.

Chapter 6 (The Para-Functional Language) introduces an annotation language (on expressions and types) by which the programmer indicates parallelism; a formal framework is developed to transform the annotations into pseudo-functional operations.

Chapter 7 (Parallel Compositions) introduces parallel variants of high-level compositional forms that implement typical algorithmic patterns and are compiled into annotations or pseudo-functional operations. The runtime behavior of these constructs is analyzed in detail.

Chapter 8 (Application Examples) presents the parallelization of several algorithms, first starting with simple benchmark programs and then continuing with non-trivial computer algebra algorithms.

Chapter 9 (Related Work) discusses related research work on systems for parallel computer algebra and parallel implementations of functional programming languages.

Chapter 10 (Future Work) discusses several directions in which our research work will be continued.

Appendix A (The Correctness Lemmas) presents and proves the auxiliary lemmas required for Chapter 3.

Appendix B (The pD Language) presents the lexical structure and the context-free grammar of pD and describes the interface to the PACLIB kernel.

Appendix C (Target Code) lists examples of (sequential and parallel) target code generated by the pD compiler.

Chapter 2

The PACLIB Runtime Kernel

We describe the pD system bottom-up starting with the runtime kernel that is in charge of the automatic task and memory management. This runtime system is actually the shared memory implementation of an imperative parallel programming model that was designed for the development of a parallel version of the computer algebra library SACLIB [Buchberger *et al.*, 1992]. Because of this origin, the kernel design had to meet several constraints that are described in Section 2.1.

Already from the very beginning the intention was to use this kernel later as the runtime system of a parallel functional programming language. Parallel computer algebra algorithms could be then developed in a higher-level programming style but would be compatible to the already existing sequential and parallel software. This long-term goal heavily influenced the basic task creation, communication, and synchronization model.

However, the target code generated by the pD compiler is not bound to the PACLIB kernel at all. It can be easily linked with any runtime system that supports

1. **Garbage Collection.** The target code is transparent to memory management. The allocation of basic objects (lists, tuples) must be supported.
2. **Task Creation.** Parallel tasks may be created to execute function applications. Tasks are first order entities that can be used like any other objects.
3. **Task Synchronization.** The result of a task may be retrieved without destroying the task. Hence, multiple synchronizations on a task are allowed.
4. **Non-Determinism.** If the non-deterministic features of pD are applied, the kernel must also support non-deterministic synchronization and task abortion.
5. **Forward-Declared Tasks**. If cyclic data structures are used in pD programs, task allocation must be separated from task initialization/activation.

In fact, while this kernel model is based on the imperative programming language C, it is functional in spirit: tasks communicate essentially by function arguments and results. Consequently, the model can be implemented without major change on distributed memory hardware as well. The PACLIB kernel is also the essential substrate of the multi-processor version of the new C++ based computer-algebra library STURM [Hong *et al.*, 1994b]. For this variant of the kernel, a PVM-based cluster port is currently under way [Geist and Sunderam, 1992, Bertoli *et al.*, 1994].

An overview of the PACLIB kernel is given in [Schreiner, 1992, Schreiner and Hong, 1993c]; the virtual task management in Section 2.6 was published in [Schreiner, 1994c].

2.1 Design Goals

In the design phase of the PACLIB kernel, several demands were raised that the parallel computer algebra kernel should fulfill:

1. **Upward compatibility to the SACLIB kernel:** The PACLIB kernel should provide the same interface to the heap and the same efficiency of heap access as the SACLIB kernel. Each function of the SACLIB library should therefore run without change and without loss of speed when linked with the PACLIB kernel.
2. **High-level parallel programming model:** The system should be based on a parallel programming model that is simple to use. It should be possible to spawn each function of the SACLIB library as a concurrent task and to retrieve the result of this function without any change of the interface. Tasks should be first-class objects.
3. **Light-weight concurrency:** Since many parallel algebraic algorithms are rather fine-grained, the runtime overhead for the creation of a new task should be very small and it should be possible to spawn thousands of tasks.
4. **Non-determinism and speculative parallelism:** Since the runtime of algebraic algorithms can often not be predicted, there should be a synchronization construct waiting for a whole set of tasks and returning the first available result. If the results of the other tasks are not needed any more, it should be possible to abort these tasks.
5. **Communication by heap references:** On a shared memory machine, task arguments and task results should be heap references only. Neither the creation of a new task nor its termination should require to copy whole SACLIB structures (which would increase the runtime and might duplicate originally shared substructures).
6. **Parallel garbage collection:** There should be a global garbage collector that on demand reclaims all of the no more referenced heap cells. The garbage collector should be parallelized in order to profit from the multi-processor architecture.

In the following subsections, we will describe how and to which extent we met these demands.

2.2 The Kernel Interface

Most algebraic algorithms are based on purely mathematical functions that are entirely defined by their argument/result behavior. The basic PACLIB model of parallelism reflects this view. A PACLIB task receives certain SACLIB objects as input arguments and returns a SACLIB object as its result. This result may be retrieved later by a handle to this task.

t = `pacStart(`f`, `a_i`)`

creates a new task that asynchronously executes $f(a_i)$. The task handle t is a first-order object that can be passed to other functions/tasks and stored in any SACLIB data structure. The workspace (stack) of the task is immediately allocated and the task is put into the ready queue for execution (eager task creation).

v = `pacWait(&`t`, `ts`)`

returns the result v of one of the denoted tasks ts and returns the handle t of the delivering task. If all denoted tasks are still active, `pacWait` blocks until a task terminates.

```
Word PIPROD(Word a, Word b) {          /* parallel integer product */
  Word l, n, i, j, b0, b1;
  Word t, tlist;
  l = LENGTH(b);                       /* length of b */
  n = l / N + (l % N != 0);            /* number of tasks */
  tlist = NIL;                         /* list of tasks */
  for (i = 0; i <= n; i ++) {          /* start tasks */
    ADVANCE(N, b, &b0, &b1);           /* split off subinteger b0 */
    j = i*N;                           /* shifting factor */
    t = pacStart(IPRODS, 3, a, b0, j); /* start task for (a*b0)<<j */
    tlist = COMP(t, tlist);            /* put task into list */
    b = b1;                            /* rest of b (without b0) */
  }
  for (i = 0; i < n-1; i++) {          /* collect results */
    b0 = pacWaitListRm(&tlist);        /* 1st result, remove task */
    b1 = pacWaitListRm(&tlist);        /* 2nd result, remove task */
    t = pacStart(ISUM, 2, b0, b1);     /* start task for b0+b1 */
    tlist = COMP(t, tlist);            /* put new task into list */
  }
  return(pacWaitOne(FIRST(tlist)));    /* result of last task */
}
```

Figure 2.1: Parallel Integer Multiplication

`pacWait` is a *non-deterministic* construct whose result in general depends on the system situation at runtime. Only when applied to a single task, its result is uniquely determined. The additional level of flexibility provided by non-determinism may help to considerably reduce the number of task blockings.

`pacWait` is also *non-destructive* i.e. it does not destroy the task whose result is delivered. Actually the workspace of a task t is reclaimed immediately when the task terminates. The task result is reclaimed by the garbage collector when t is not referenced any more. As a consequence arbitrarily many tasks may asynchronously wait for the same set ts without interference. This feature distinguishes the PACLIB kernel from most other thread packages except for the MultiLisp futures [Halstead, Jr., 1985].

There exist several variations of `pacWait`: `pacWaitListRm` takes as its single argument a SACLIB list of task handles and removes from the list the task whose result is delivered; `pacWaitOne` waits for the result of a single task.

`pacStop(`*tasks*`)`

terminates the execution of all denoted *tasks* and of all subtasks created by these tasks. If some task has already delivered its result, it is not affected any more. `pacStop` can be considered as an annotation that does not influence the correctness of a program (modulo termination) but only its operational behavior.

t = `pacVirtual(`f,a_i`)`

creates a new task description but does not yet allocate a workspace for it. Consequently the task is not yet prepared for execution and just put into a "virtual" ready queue (lazy task creation). See Section 2.6 for more details.

t = `pacForward()`

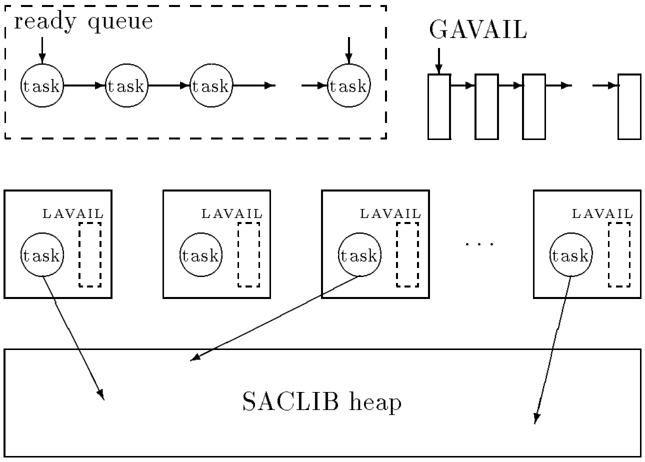


Figure 2.2: PACLIB Kernel Design

returns a task handle *without* associated task. Each of the calls

```
pacStartFwd(t, f, a_i)
pacVirtualFwd(t, f, a_i)
```

takes such a handle t as its argument and creates a (virtual) task whose result can be retrieved by this handle. This "forward declaration" is required in the context of circular programs where tasks may mutually recursively wait for their results (see Section 6.4.5).

Figure 2.1 illustrates the PACLIB model of parallelism by a parallel version of the school algorithm for the multiplication of two integers a and b. `PIPROD` splits b into subintegers b_0 of N digits and multiplies a with each b_0 in parallel. Each task executes the function `IPRODS` that multiplies a with b_0 and shifts the result by j digits. `PIPROD` iteratively waits for the results delivered by any two tasks and starts a new task that adds the results. Finally, there remains a single task left that returns the total result.

There are several parallel algebraic algorithms (e.g. for *critical pair completion*) for which the purely functional model is less adequate. These algorithms can be better described in terms of tasks each of which iteratively receives a sequence of input objects, processes each object according to its internal state (which is changed by the computation) and forwards a sequence of results to other tasks. Therefore PACLIB supports an additional form of task communication by *streams* i.e. automatically synchronized lists that may be used for this purpose. We omit the details since the pD language implements its own notion of streams (see Chapter 5).

For a comprehensive description of the PACLIB programming model and its application, see [Hong *et al.*, 1992].

2.3 The Kernel Design

The task management of the PACLIB kernel is built on top of the μSystem [Buhr and Stroobosscher, 1990]. This free software package consists of a library of C functions that supports light-weight concurrency on shared memory multiprocessors and workstations under Unix.

The μSystem kernel distributes *tasks* (light-weight processes) among *virtual processors* which are (heavy-weight) Unix processes scheduled by the operating system. On a multiprocessor, virtual processors are therefore executed by multiple hardware processors truly in parallel. The μSystem task management is quite efficient and allows the utilization of rather *fine-grained parallelism*.

The PACLIB kernel consists of the following main components (depicted in Figure 2.2):

1. **Ready Queue:** This queue contains all tasks that are *active* but not *running*. The scheduler selects tasks for execution from the head of this queue; tasks that are preemptively descheduled are inserted at the tail.
2. **Shared Heap:** The SACLIB heap serves as the only communication medium between tasks. There is no communication bottleneck since all PACLIB tasks may read all heap cells simultaneously.
3. **Virtual Processors:** Each virtual processor is a Unix process that owns a local list `LAVAIL` of free heap cells. Each task executed on this processor allocates new heap cells from this list.
4. **Global Available List:** `GAVAIL` is a global list of free lists in which the unused heap cells are linked together. Processors whose local lists run out of free cells receive their new `LAVAIL` from `GAVAIL`.

The PACLIB kernel extends the μSystem task management by an original result delivery scheme where the termination of a task is decoupled from the delivery of a result. Consequently, a task may deallocate its workspace when it finishes execution without waiting for another task to retrieve the result. Furthermore, multiple tasks may retrieve the result of the same task without interference. This management of tasks is described in Section 2.4.

Moreover, the PACLIB kernel implements an additional level of abstraction by *virtual tasks* that are scheduled among tasks like tasks are scheduled among processes. Consequently, finer-grained parallelism is possible but in particular memory requirements are drastically reduced. Virtual tasks were introduced by [Mohr *et al.*, 1990, Mohr *et al.*, 1991] and [Küchlin and Ward, 1992]; Section 2.6 describes our refinements that were also published in [Schreiner, 1994c].

2.4 Task Management

We applied and considerably extended the features of the μSystemin order to implement the PACLIB parallel programming model which is higher-level and more general than the native one. There were two major problems that had to be solved:

1. How to represent the task handles returned by `pacStart` and
2. How to implement the non-deterministic functionality of `pacWait`.

For implementing task handles, there basically exist two possibilities: The first one is to view such a handle as a *task descriptor* i.e. as a reference to the workspace of the task. Before a task terminates, it stores its result in this workspace for the later retrieval by other tasks. Then either the workspace is not reclaimed as long as there are references to the task or the workspace is reclaimed when the result is received by another task. However, the first approach would waste too much space while in the second approach (implemented in the μSystem) tasks are not first-order objects any more.

Therefore we decided to split the description of a task into two separate entities (see Figure 2.3):

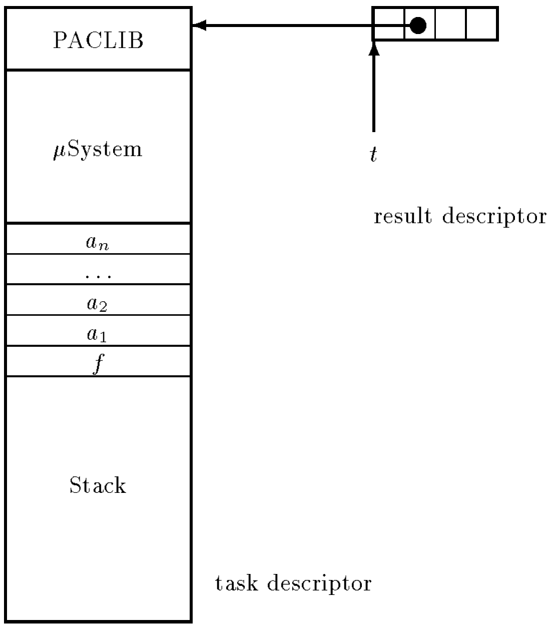


Figure 2.3: A PACLIB Task

1. The *result descriptor* is a small structure that serves as the rendezvous and synchronization point to other tasks.
2. The *task descriptor* is a small extension of the μSystem task descriptor that is placed on top of the task stack (32 KB by default).

A PACLIB *task* is actually a reference to the result descriptor which is a SACLIB list with the following elements:

`sem`	`task`	`wait`	`value`

- `sem` is a (pointer to a) semaphore that provides mutual exclusion on the descriptor.
- `task` is a pointer to the task descriptor of the computing task. If the task has already terminated, `task` is the constant `AVAILABLE`.
- `wait` contains, if `task` ≠ `AVAILABLE`, the head of a queue of tasks blocked on an attempt to receive the result.
- `value` contains, if `task` ≠ `AVAILABLE`, the tail of the `wait` queue and the result of the task, otherwise.

A *task descriptor* is a record that is placed on top of the stack of the task and contains (among others) the following fields:

`sem`	
`prev`	`next`
`result`	
`value`	
`wait`	
`sub`	
`stack`	`stop`

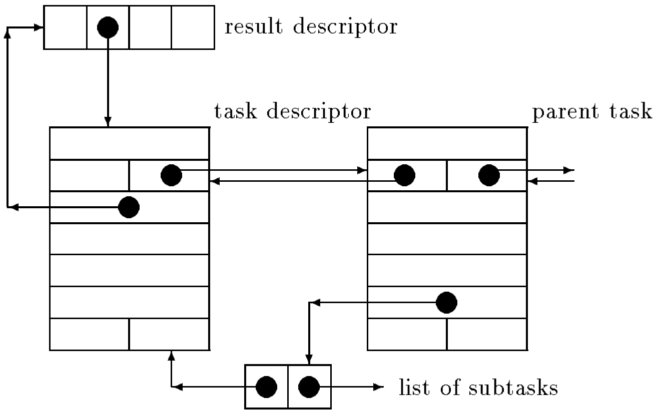


Figure 2.4: Task Creation

- `sem` is a (pointer to a) semaphore that provides mutual exclusion on the descriptor.
- `prev` and `next` connect all existing task descriptors into a doubly linked list.
- `result` is a reference to the result descriptor of the task.
- `value` is a location that receives the result of another task if the current task is blocked in a call of `pacWait`.
- `wait` contains, if the task is blocked in a call of `pacWait`, a list of those result descriptors that the current task is waiting for.
- `sub` is a list of result descriptors of all subtasks created by the current task.
- `stack` is an integer that denotes the stack size to be allocated for all tasks created by the current task.
- `stop` is a flag that is set if another task wants to abort the current task.

`t = pacStart(`f`,`a_i`)` allocates a task descriptor and stores the function pointer f and the arguments a_i at the begin of the stack. A result descriptor is allocated and mutually linked with the task descriptor a reference to this result descriptor is returned as the task handle t (see Figure 2.4).

The new task executes a function `pacShell` that calls $f(a_i)$. When f has returned its result, `pacShell` stores the value in the result descriptor and awakes all tasks blocked on this descriptor. Finally `pacShell` erases the task link in the result descriptor, deallocates the task descriptor and terminates the task.

The large task descriptor therefore occupies space only as long as the task is executing. After termination only the small result descriptor remains active from where subsequent `pacWait` calls will return the result without blocking. Since this descriptor is a SACLIB object, it is subject to the general garbage collection mechanism and together with the result value reclaimed when no task has the handle t any more. Hence, also task results do not occupy memory longer than necessary.

The non-deterministic nature of `pacWait` actually requires a complex result delivery protocol that runs roughly as follows (the numbers in Figure 2.5 refer to the steps of the algorithm):

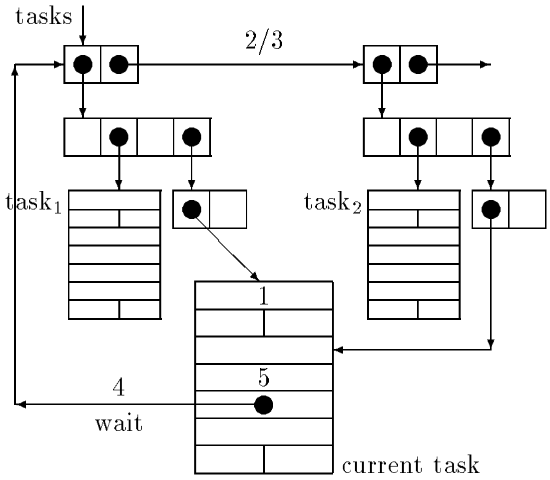


Figure 2.5: Waiting for a Result

1. **Lock:** The semaphore associated to the descriptor t of the current task is locked. This prevents the *tasks* from delivering their result to t before `pacWait` has reestablished a consistent state.

2. **Scan:** All result descriptors in *tasks* are scanned and the result descriptor r of the current task is linked into the waiting queues of these descriptors.

3. **Found:** If a result descriptor in *tasks* already contains a result, r is removed from all waiting queues it has previously been linked into. t is unlocked and the result value is returned.

4. **Sleep:** If none of the results is already available, the result descriptor list *tasks* is stored in the `wait` field of t. t is unlocked and the current task is blocked.

5. **Awake:** After the task has been awaked, the `value` field of t contains the result of one of the *tasks* it waited for. Likewise, the `wait` field now contains the result descriptor of the task that returned the result.

After the `pacShell` has received the result of *fun*, it forwards the value to all tasks waiting on the result descriptor r (see Figure 2.6):

1. **Pick Task:** The semaphore associated to r is locked and the first task t waiting in the queue is determined. In order to prevent a deadlock situation, r is unlocked before t is locked and then locked again. Without this detour, a rival task delivering its result to t might already have locked t and itself try to lock r in order to remove t from the waiting queue of r.

2. **Check Task:** If t has disappeared from the waiting queue of r, the corresponding task has in the meantime received its result from a slightly faster rival task. Otherwise, the current task has "won" the race and may deliver its result to t.

3. **Remove Descriptor:** t is removed from the waiting queue of r and from the waiting queues of all other result descriptors that the task is waiting for (the `wait` field of t holds a list of these descriptors). This guarantees that only one result is delivered to t.

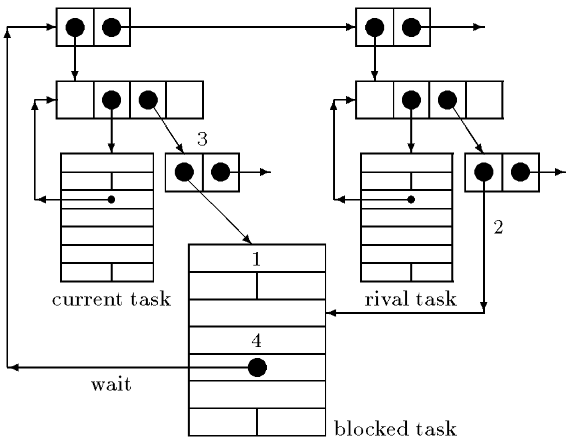


Figure 2.6: Delivering the Result

4. **Deliver Result:** The result value is written into the `value` field of t and r is written into the `wait` field of t. The corresponding task is then awaked and can extract the desired information from t.

After the result has been delivered to all waiting tasks, it is written into the `value` field of r. The `task` field of r is set to `AVAILABLE` and r is unlocked. The task then unlinks the task descriptor from the global task list, releases its workspace and terminates.

This protocol for the delivery of task results contains a few non-trivial details; Chapter 3 presents a formal verification of the algorithm using a temporal logic framework [Schreiner, 1993c].

2.5 Heap Management

The PACLIB kernel provides an operation `COMP` with the same interface and efficiency as the SACLIB kernel. However, the organization of free lists in the PACLIB kernel is much more sophisticated: There is a free list `LAVAIL` attached to each (virtual) processor and there is a global list of free lists `GAVAIL`. When a task executes `COMP`, the new cell is allocated from the `LAVAIL` of that processor that currently executes this task. If `LAVAIL` runs out of free cells, the processor picks the first list in `GAVAIL` as its new `LAVAIL`. If `GAVAIL` is also empty, a global garbage collection is triggered. This two-level memory management scheme of PACLIB is a compromise between two extremes:

1. **Global Available List Only:** In order to keep `GAVAIL` consistent, it must not be simultaneously updated by different processors. Without local available lists, each call of `COMP` would require a semaphore operation that would considerably increase the runtime overhead of the function. Moreover, due to the serial access to `GAVAIL`, this shared resource would represent a significant bottleneck for parallelization.

2. **Local Available Lists Only:** Without a global available list, there arises the danger of wasting heap space. Some processor might consume cells from its local available list `LAVAIL` faster than the other processors. It would then trigger garbage collection

even if there were still plenty of free cells available (since these cells were attached to the local available lists of other processors).

In order to trigger garbage collection, the processor puts the currently executed task back into the ready queue and interrupts by a Unix signal the execution of the other (virtual) processors. After all processors have acknowledged the interruption, garbage collection takes place in two phases:

1. **Mark:** All cells that can be referenced by any task are marked. Since all user tasks are descheduled, all heap references are on the stacks of these tasks. Each virtual processor picks a task descriptor, scans the associated stack and marks all heap cells that can be reached.

2. **Sweep:** All unmarked cells are reclaimed to construct a new list `GAVAIL`. Each processor scans a different portion of the heap, unmarks the marked cells and reclaims the unmarked cell. When a new free list of the desired length is built, the list is linked into `GAVAIL`.

The work of the sweep phase is well balanced among all processors since they operate on different heap sections of the same size. The balance of load in the mark phase depends on the number of available tasks and the number of cells that can be reached by each task. The experiments presented in Section 2.7 show that the efficiency of the garbage collector is in general very high.

2.6 Virtual Tasks

2.6.1 Motivation

In PACLIB parallelism may be utilized at a rather low-level yielding a large number of tasks with small grain-sizes. However we (as other researchers before us) soon faced an obvious discrepancy. Parallel program development is considerably simplified by the ability to create a large number of light-weight tasks, but each task requires a significant amount of system resources (in particular memory) and is subject to some performance penalty.

Since the number of physical processors is actually very limited (20 on our system), the large number of light-weight processes is not appropriately rewarded by reducing the computation time at the same scale. In practice, during the fine-tuning of a parallel program the number of tasks is therefore reduced at least to the same order of magnitude (say some 100) as the number of processors.

The techniques of *lazy task creation* [Mohr *et al.*, 1990, Mohr *et al.*, 1991] or *virtual threads* [Küchlin and Ward, 1992] were developed to overcome this discrepancy. Virtual threads are basically just descriptions of threads that may be created at a low cost but are themselves not yet executable. The idea is that most of these virtual threads will never become real at their own (thus avoiding the overhead) but their descriptions will be executed by other real threads that are already in existence. Consequently, the number of real threads will be reduced and their grain size will be increased.

In the following subsection we describe the concept of virtual tasks as implemented in the PACLIB kernel. This implementation was inspired by the virtual S-thread package of [Küchlin and Ward, 1992] but also contains a variety of original features [Schreiner, 1994c].

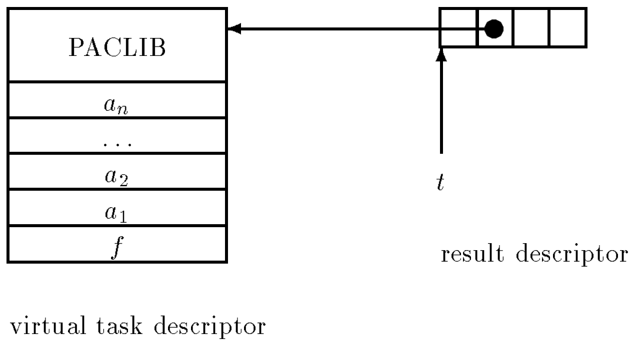


Figure 2.7: A Virtual PACLIB Task

2.6.2 Implementation

Virtual PACLIB tasks are created by a function call

t = `pacVirtual(`f`,` a_i`)`

with the same interface and basic semantics as a `pacStart` call. The difference between both calls is the activation time of the task t: a call of `pacStart` immediately creates a full task descriptor and links it into the ready queue of tasks to be selected for execution.

`pacVirtual` however only allocates a virtual task descriptor as depicted in Figure 2.7. This descriptor consists of the PACLIB extension of the μSystem task descriptor and space for the function pointer f and arguments a_i. Virtual task descriptors have fixed size (about 80 bytes) and are obtained by locked access from a central pool (that is dynamically extended on demand). The descriptor is linked into a "virtual" scheduling queue that is distinct from the ready queue of "real" tasks.

The new result descriptor contains a link to the virtual task descriptor; the handle t returned by `pacVirtual` may participate in all operations like the handles of result descriptors that are returned by `pacStart` and thus connected to real tasks.

A virtual task is "realized" (i.e. transformed into a real task) on the following occasions:

1. **Idling:** The scheduler of a virtual processor finds the global queue of executable tasks empty. In this case it picks a descriptor from the virtual task queue and allocates a full-sized task descriptor with a stack. The contents of the virtual descriptor are transferred into the task descriptor and the link of the corresponding result descriptor is reset to this descriptor. The virtual descriptor is freed and the new task is activated.

2. **Termination:** A task is going to terminate after it has delivered its result. In this case the task tries to grip a descriptor from the virtual task queue. If such a virtual task is found, the task overwrites its own descriptor with the retrieved information, resets the link in the corresponding result descriptor, frees the virtual task descriptor and restarts its execution with the computation of the denoted function.

3. **Deterministic Wait:** A task calls `pacWait` for a single task that is still virtual. In this case, the task grips the virtual task, computes the denoted function itself, delivers its result into the corresponding result descriptor (and to all tasks waiting there) and continues execution.

4. **Non-deterministic Wait:** A task calls `pacWait` for several tasks some of which are still virtual. In this case, the task realizes all virtual tasks before it gets blocked. Otherwise, the semantics of `pacWait` would be effectively changed, since some of the virtual tasks might never become active (if other tasks do not terminate).

Case 1 is handled by the scheduler code executed in every virtual processor. Case 2 is managed by the `pacShell` function that every PACLIB task executes. Cases 3 and 4 are covered by the `pacWait` function. Consequently no change in the user code is necessary, i.e. the management details of virtual tasks are entirely hidden from the user. The extra runtime overhead in `pacShell` and `pacWait` consists of a simple test operation.

2.6.3 Consequences

We will now discuss some of the consequences of our scheduling scheme for virtual tasks:

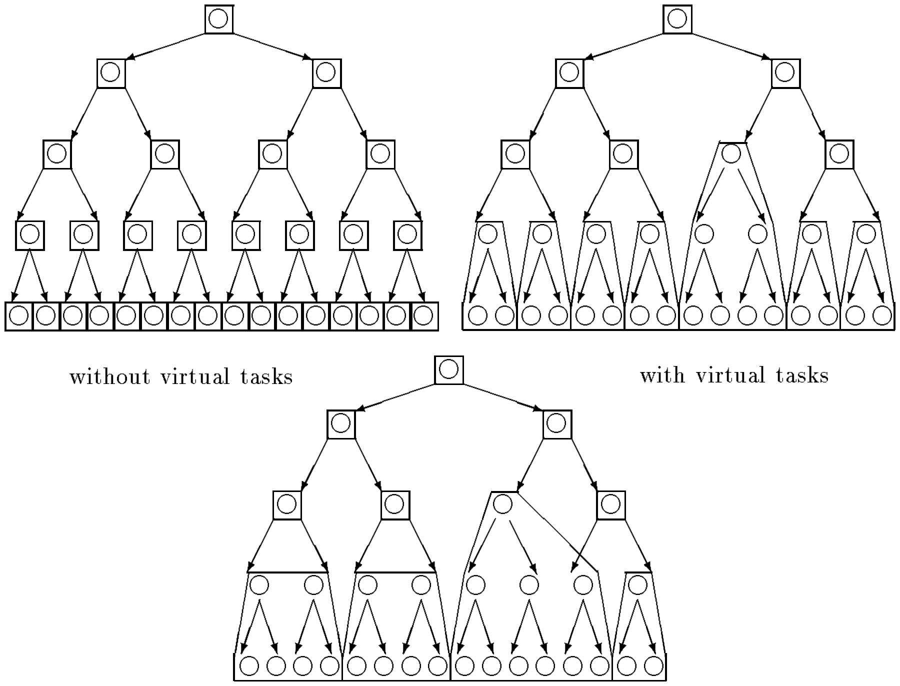
1. A real task is only created (unless explicitly by `pacStart`) when a processor would idle (or the semantics of the program would be changed) otherwise. The application of virtual tasks may therefore essentially reduce (a) the total number of real tasks ever created and (b) the maximum number of real tasks that exist at any moment.
2. Virtual tasks have lower priority than real tasks. While the available processor time is fairly scheduled among all real tasks on a preemptive basis, a virtual task becomes only active when some real task or some processor has no other choice (in particular if there is nothing else to do).
3. A task is only terminated when there is no more virtual task to be executed, hence the workspace of a task is reused as long as possible. This may essentially improve the data locality of a program by the reduction of its overall memory requirements (yielding a higher cache hit ratio and less disk paging).
4. In a deterministic wait, a task is only blocked when the task connected to the result descriptor is currently active. Otherwise, the corresponding task either has already delivered the result or is still virtual and gets "inlined" into the current task.
5. In a non-deterministic wait, none of the participating tasks is virtual. This would effectively change the behavior of the program since some of the virtual tasks might never (or at least too late) become active.

Since a task may be blocked only on a `pacWait`, the last two arguments show that virtual tasks do not change the termination semantics of a PACLIB program. Under the assumption that all tasks eventually terminate, we may even lift the last restriction on the non-deterministic case without harm. However, it is questionable whether it is desirable to wait non-deterministically on tasks that have not yet been activated.

2.6.4 Comparison

While there are several similarities, our concept differs in some important aspects from Küchlin and Ward's scheme. In their package virtual threads are transformed into real threads on two occasions:

1. When executing a `vthread_join(`t`)` operation (a deterministic destructive wait), the current thread checks whether t is still virtual. If this is the case, the current thread executes the virtual thread itself.



without virtual tasks with virtual tasks

with task reuse

Figure 2.8: A Divide-and-Conquer Tree

2. A scheduler thread wakes up from time to time to check the current load. If the number of active threads is below some threshold, the scheduler transforms an appropriate number of virtual threads into active threads. A copy of the scheduler code is also run by each thread as a side effect of a potentially load changing virtual thread call.

When a thread is going to terminate, this scheme does not reuse the thread for the execution of another virtual thread but the workspace is effectively deallocated. New real threads are later created when the scheduler decides that the current load is too low.

In our scheme tasks are never terminated if there still exist virtual threads to be executed. Furthermore we do not use a global load estimation scheme to start new real tasks. Instead every processor itself transforms virtual threads into real threads if it would idle otherwise.

A consequence of the different task scheduling policies is depicted in Figure 2.8 illustrating a divide-and-conquer algorithm where each task recursively creates two new tasks:

1. Without using virtual tasks, a real task is created for each node in the call tree; all non-leaf tasks are blocked waiting for the results of their children.
2. In Küchlin's and Ward's scheme, a single real task will typically execute a whole subtree (by inlining the corresponding virtual tasks); when this subtree has been processed, the task terminates.
3. In our scheme, a single real task may execute independent subtrees; when one subtree is processed (by inlining the corresponding virtual tasks), the terminating task will execute any other unprocessed subtree.

The distinction between both schemes becomes more compelling for linearly recursive functions of the type

```
f(x) = if (x == 0) then b else h(g(x),f(x-1))
```

We assume that on a call $f(n)$ the current task recursively starts n virtual tasks for the execution of the $g(x_i)$. In Küchlin and Ward's scheme, then $O(n)$ real tasks are created since each real task computing some $g(x_i)$ cannot inline any other virtual task. In our scheme, $O(P)$ real tasks are created (where P is number of processors) since each real task that has finished the computation of some $g(x_i)$ may continue with the computation of any other $g(x_j)$.

2.6.5 Advantages

What exactly are the main advantages of virtual tasks? As emphasized by [Küchlin and Ward, 1992], virtual task reduce the task management overhead and thus allow finer-grained parallelism. The results in Section 2.7 support this observation but also demonstrate that this reduction is not large enough to yield a really new quality of parallel programming.

However, virtual tasks have another advantage that provides such a new quality: By the delayed creation of tasks, the maximum memory requirements of a program may be extremely reduced. Consequently, some programs may execute with virtual tasks that are not able to execute with real tasks. Moreover, due to virtual memory management of the operating system, many programs are extremely slowed down by their large memory consumption; these programs may run much faster with virtual tasks.

We will demonstrate this on the example of a divide and conquer parallel algorithm.

Application of Real Tasks

A divide and conquer algorithm usually provides much more parallelism than physical processors are actually available. If we assume that in each level of the call tree one of the recursive calls is spawned as a real task while the other one is executed within the current task, a call tree of height n generates $N = 2^n$ real tasks in total.

We may now put the question now how many of these tasks are in existence at a time, in particular what is the *maximum* number of tasks in existence. Let us assume that we have a parallel system with $P = 2^p$ processors where $p \ll n$. Let us further assume that scheduling is done non-preemptively using a FIFO queue for real tasks (no virtual tasks are used).

The main task recursively spins down the left-most path of the call tree starting n tasks for the right subtrees rooted in this path. These children behave like their parent by spinning down their left-most call paths and starting new tasks for their right subtrees. However, since we have only P processors, only the tasks started in the uppermost p tree levels find a processor for execution while the others only reside in the ready queue. Consequently only the oldest p of the n children of the main task are actually executing.

Now take a look at the behavior of one of the P executing tasks in charge of a subtree of height $m \leq n - p$. After the task has created m children, it solves its base case, steps one level up and gets blocked when it demands the result of its youngest child. However due to the FIFO scheduling the $m - 1$ elder sisters get scheduled before this child can execute and release its parent. The oldest child starts $m - 1$ tasks before it gets blocked, the second

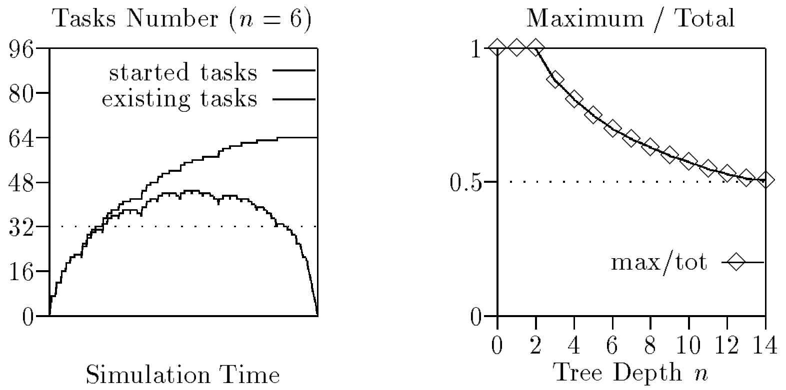


Figure 2.9: Simulation Results

oldest child starts $m-2$ tasks, and so on. Before the youngest child may execute, release its parent and terminate, $\sum_{i=1}^{m} i = O(m^2)$ tasks have been created.

The parent task then combines its own subresult with the result of the terminated child, steps one level up and gets blocked again. Now all of the $O(m^2)$ existing tasks are scheduled creating $\sum_{i=1}^{O(m^2)} i = O(m^3)$ new tasks. After a few such rounds already a significant fraction of the total number of tasks has been created while only a few leaf tasks (and few tasks close to leaves) have terminated. The FIFO scheduling strategy effectively favors the breadth-first task expansion of the call tree over the termination of leaf tasks.

The left picture in Figure 2.9 shows the results of a simulation of the FIFO strategy for a task executing a tree of height $n = 6$ with a single processor, which corresponds to the execution of one of the P task trees. One can clearly see on the left ascent the inverted m^2 curves resulting from the creation of $m, m-1, \ldots, 1$ tasks separated by small cuts that denote the termination of a task. During most of the time more than 50% of the total tasks are in existence. The right picture shows that for increasing tree height the ratio between the number of existing tasks at a time to the total number of tasks converges to 0.5 i.e. the maximum task number is $O(2^n)$.

Application of Virtual Tasks

What changes if we apply virtual tasks? Initially all virtual tasks are realized by idle processors. After saturation, each of the P real tasks inlines its virtual children in the order in which their results are required (youngest first) i.e. the call tree is processed depth first. Since this actually leads to a LIFO scheduling strategy for virtual tasks, the maximum number of virtual tasks is $O(n)$. On the other hand the maximum number of real tasks does *not* depend on the tree height n. If the call tree of all P tasks were of equal height, no more than P real tasks would

In fact the realization of virtual tasks by idle processors takes considerably more time than the creation of real tasks. During any moment t before saturation, the number of real tasks N_t is not $2N_{t-1}$ (assuming unit time for the creation of a virtual task) but $N_{t-1} + N_{t-c}$ where c is the time required for the realization of a virtual task. Until all processors become saturated a virtual task has therefore some time advantage over its “nieces” (the children of its elder sisters) for getting realized. The call trees of the initial P real tasks are therefore rather unbalanced and the corresponding tasks complete execution at different times.

Every real task that has completely processed its call tree grips the first virtual task in the queue (which is the oldest child of some other real tas) for further execution. The parent of the gripped task will be blocked later (because a task takes for processing its part of the call tree no longer than its oldest child). Since initially any blocked task causes some processor to idle, an additional virtual task is realized and more than P real tasks come into existence. Still the maximum number of real tasks is $O(P)$ since after a few iterations of task blockings and realizations every processor whose task becomes blocked finds already some other real task in the ready queue.

Why Reducing the Number of Tasks?

üchlin and Ward's original motivation for the introduction of virtual tasks was to overcome a limitation of their operating system that allowed only a small number of real tasks. In the PACLIB system such a limitation does not exist. However we consider the reduction of the number of real tasks at a time as *the* advantage of virtual tasks which is of essential importance for the *runtime* of a PACLIB program and for its *applicability* in general.

On operating systems with virtual memory support the runtime of a program also heavily depends on its memory consumption and data locality. The PACLIB system equips a task by default with a 32 KB stack. The user may reduce this size but it will be difficult for a task to run with less space without risking stack overflow for recursive functions. Consequently 1000 real tasks already require 32 MB, 10000 tasks will defeat the system. This large memory consumption together with regular context switches (by the preemptive scheduler) lead to a very low data locality in time and space.

First the cache hit ratio becomes rather low. Our Sequent Symmetry provides 512 KB cache memory per processor which suffices to keep the stacks of say 10 tasks in cache (we also have to consider the accesses to program text and heap memory). Context switches to additional tasks will lead to drastically increased cache misses and consequently to heavy bus traffic.

For more than some hundred tasks, the memory consumption becomes so large that paging will set in: the operating system will start to transfer virtual memory pages that contain stacks of recently not used tasks to the disk. Since each page fault requires say 10 ms (the equivalent of roughly 20000 function calls), the performance of the program will drastically degrade. It is illusory to start on a machine with 128 MB of main memory 10000 real tasks.

The situation changes if we apply virtual tasks; in the best case only as much real tasks exist at any time as physical processors. Even if somewhat more real tasks have to be created (due to other tasks being blocked), the virtual task mechanism in general allows us to start 10000 virtual tasks and still keep the memory consumption low. Furthermore, since the inlining of virtual tasks may lead to a FIFO scheduling strategy, the fraction of heap memory active at a time may reduce and make garbage collection more effective.

2.7 Benchmarks

We present the results of several benchmarks to illustrate the efficiency of the runtime system. The timings were performed on a lightly loaded Sequent Symmetry multi-processor with 128 MB of physical memory and 20 processors i386 running at 16 Mhz. The time

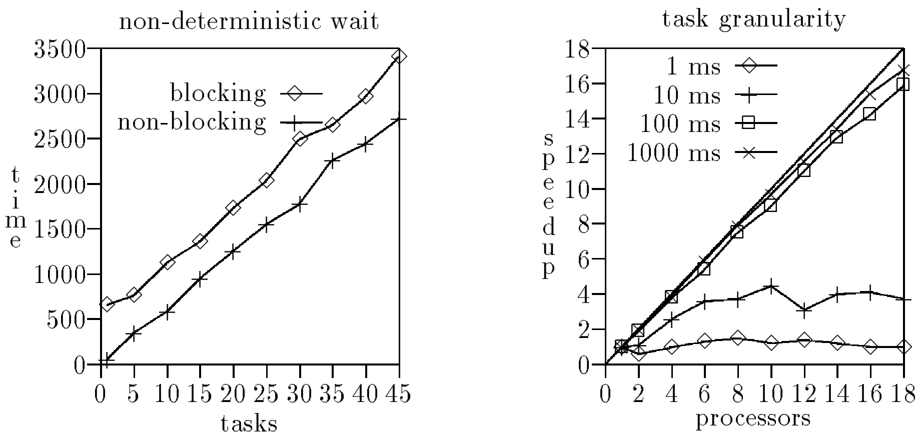


Figure 2.10: Task Benchmarks

overhead imposed by the PACLIB kernel is negligible i.e. a sequential program runs as fast as when linked with the SACLIB kernel.

2.7.1 Task Creation and Synchronization

The following table displays the time overhead of the basic PACLIB operations `pacStart` and `pacWait` once for real tasks and once for their virtual counterparts (using `pacVirtual` instead). The wait operation is performed when the result is not yet available such that the requesting task gets blocked. The figures in parentheses are normalized with respect to a function call with 4 arguments (= 1) which takes about 5 μs on our Sequent Symmetry.

	Real task	Virtual task
`pacStart`	260 μs (52)	85 μs (17)
`pacWait`	340 μs (68)	85 μs (17)

These figures illustrate that the application of virtual tasks considerably reduces the creation overhead and also the synchronization overhead. However it is obvious that this overhead is still an order of magnitude larger than the overhead for a simple function call.

In the `pacWait` benchmark, a single task argument was used (deterministic wait). Actually, the time required for the reception of a task result depends on whether the receiving task is blocked or not and how many results it waits for.

We therefore performed two benchmarks measuring the time that a task spends in pacWait with respect to the number of tasks m that it is waiting for (all tasks are real and the delivering task is the last one in the list). In the first test, the receiving task finds the result of the last task already available and is therefore not blocked. In the other test, none of the results is available yet; the current task is blocked and will be awaked by the last task in the list.

The results depicted in the left picture of Figure 2.10 show that the time t required for the delivery of a result is linear in the number of tasks m. Roughly, we have $t = m * 60 + b$ μs

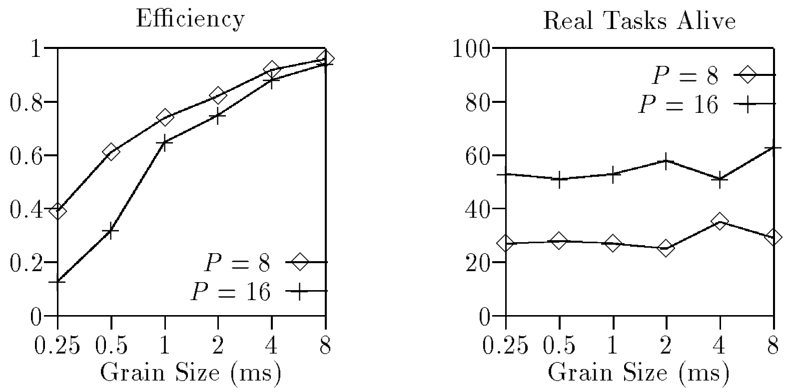


Figure 2.11: Minimum Granularity

where $b = 0$, if the receiving task is not blocked, and $b = 600$, if the task is blocked and awaked[1]. Waiting for a single task therefore costs 60 μs minimum.

Minimum Granularity and Maximum Task Number

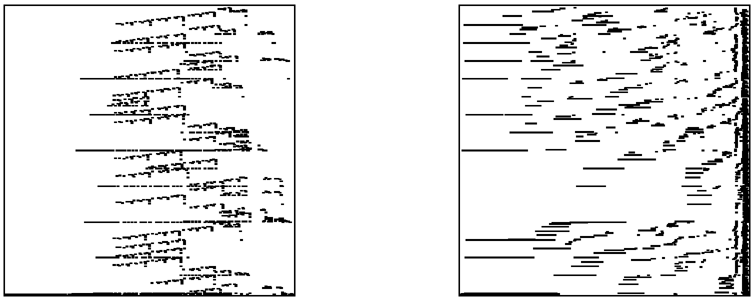
We use an artificial benchmark program to measure the minimum grain size that virtual PACLIB tasks permit. This program has a recursive divide-and-conquer structure where one recursive call is spawned as a virtual task and the task spins in the base case for a period of t ms. The program is called with $N = 12$ i.e. 4048 virtual tasks are created. Figure 2.11 displays the results of this benchmark executed with P processors:

1. The efficiency of the parallel program (compared to a corresponding sequential divide-and-conquer program) is acceptable for a grain size of $t \geq 1$ ms (which corresponds to 200 function calls). This still represents a significant constraint.
2. During the program run at most 30 respectively 60 real tasks were in existence at a time independently of the grain size t. Consequently the total memory consumption of the program was limited to about 1.5 MB (with a 32 KB stack per task).

We note that with the application of virtual tasks the number of real tasks is constant. This behavior is a consequence of the changed scheduling order of virtual tasks (depth-first) in contrast to real tasks (breadth-first) when executing a divide-and-conquer algorithm. If the time required for the realization of a virtual task were the same as for the creation of a virtual task, only P real tasks would execute at all. However, since the realization time is much longer, the generated task tree is unbalanced, which causes tasks to get blocked and additional virtual tasks to be realized to keep processors busy.

We also benchmarked a corresponding program with real tasks and a reduced stack size of 4 KB. This limited the maximum memory requirements of the program to 16 MB but still caused some disk paging. Consequently the parallel program was in all cases 50–100 times *slower* than the sequential program. These results suggest that on the Sequent Symmetry the first access to a shared memory page always causes a page fault even if there is enough physical memory available.

[1] These 600 μs include 260 μs additional cost for two context switches, since the benchmark was executed on a single virtual processor.



Virtual Tasks 246 ms Real Tasks 10219 ms

Figure 2.12: Profiling Data

Figure 2.12 visualizes profiling data generated for both program variants ($n = 10$, $t = 1$ ms) where the horizontal axis denotes the runtime. In the left diagram the lines represent virtual tasks where the long ones (actually sequences of short segments) exhibit real tasks that execute many virtual tasks. In the right diagram, lines represent real tasks; the long ones show where tasks were slowed down by disk paging (note the different time scale). This drastically shows how the memory consumption of a program also influences the runtime.

2.7.2 Garbage Collection

As described in Section 2.5, both phases of the mark and sweep garbage collector are parallelized: In the mark phase, the stacks of all tasks are investigated and the referenced cells marked in parallel; in the sweep phase, each processor scans a different section of the heap for free cells. While the amount of parallelism in the mark phase therefore depends on the number of live tasks, the work in the scan phase is always evenly distributed among all processors.

We ran a set of benchmark tests that measured the garbage collection time of the PACLIB system by a number of dummy tasks that evenly allocated heap cells. By these tests, the influence of three different parameters was investigated (see Figure 2.13):

- **Heap Size.** The program was run with heaps of size 4, 8, 16 and 32 MB. The larger the heap is, the longer the sweep phase takes (where all virtual processors can work independently in parallel). However, the total runtime differences are not very significant (using 50 tasks and a heap load of 40%).
- **Heap Load.** The amount of cells that could not be reclaimed was calibrated to 0, 20, 40 and 60% of the total heap. The more heap cells are in use, the more time the mark phase takes. For an almost empty heap, a speedup of more than 14 could be achieved compared to a speedup of 12 for a heap with 60% cells in use (using 50 tasks and a heap size of 8 MB).
- **Task Number.** The amount of tasks that allocated cells was tuned from 1 task to 25, 50, 100 and 200 tasks (with an 8 MB heap that was more than 40% loaded). The more tasks are active, the better the load balancing in the mark phase becomes. For a single task, only a speedup of 3 could be achieved; in all other cases a maximum speedup of 13 was possible.

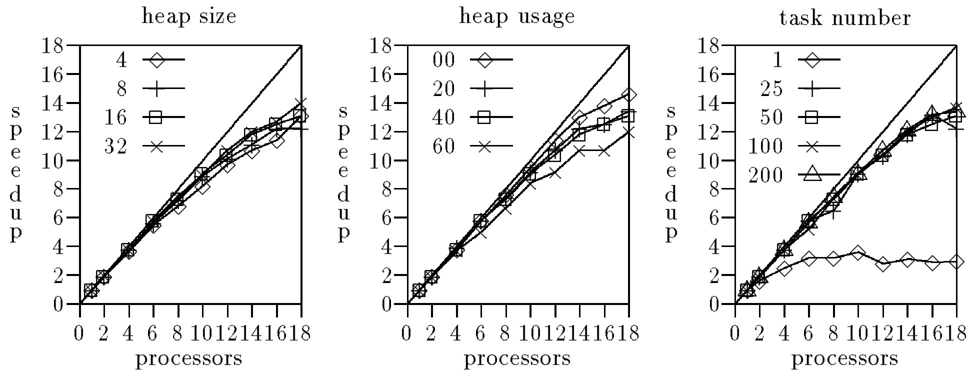


Figure 2.13: Garbage Collection

Summarizing, provided that there are sufficient tasks to saturate all processors in the mark phase, the speedup is linear up to 10 processors and achieves a maximum of 14 with 18 processors. Each processor reclaims 1.5 MB of heap memory per second which is identical to the sequential SACLIB garbage collection rate. For typical applications, the garbage collection overhead is less than 15% of the total execution time.

Chapter 3

The Correctness of the PACLIB Kernel

In Section 2.4 we sketched the implementation of the non-deterministic `pacWait` construct by a scheme whose correctness was far from obvious. We have taken the design of this protocol as an opportunity for a formal program verification applying the framework of *temporal logic* [Kröger, 1987]. This attempt has yielded interesting insights into the nature of parallel program verification and the deficiencies of the chosen methodology. Consequently, the verification was extended to a more general case study [Schreiner, 1993c].

3.1 Motivation

The verification of parallel programs is a complicated and tedious task. The more freedom a programming language provides for inter-process communication and synchronization, the more complex the interaction between parallel processes may become, the more error-prone a program is and the more issues a verification has to consider. One of the most difficult cases is that of a program where many processes are tightly coupled by shared memory and synchronized by semaphores or equivalent constructs.

High-level parallel programming languages therefore offer more restricted forms of communication and synchronization that yield less error-prone programs whose correctness may be verified more easily. In particular, the *application* of the `pacWait` model of parallelism yields programs that are quite loosely coupled; their verification is in general not very complex but can be essentially reduced to the correctness of the corresponding sequential program.

However, the *implementation* of `pacWait` is based on extremely low-level concepts; its verification is therefore a difficult task in its own. One reason for this is that the correctness of a language construct has to be established for *every* program that applies the construct. Furthermore, the implementation demands a complicated interaction of tightly coupled parallel processes. The `pacWait` algorithm is relatively short, but its dynamical behavior makes a verification difficult.

There exist several calculi for the formal verification of parallel programs e.g. the axiomatic model by [Owicki and Gries, 1976, Owicki and Lamport, 1982], the Unity model [Chandy and Misra, 1988], Milner's CCS [Milner, 1980], the Π-calculus [Milner, 1993], Hoare's CSP [Hoare, 1985] or temporal logic [Manna and Pnueli, 1981a, Manna and Pnueli, 1981b, Kröger, 1987]. The introduction of these calculi usually consists of the proof of their mathematical soundness and completeness and their application to small example programs.

However, techniques that may be useful for the verification of small programs may fail for more realistic examples. The reason for this are less mathematical deficiencies than practical ones, for instance the complexity of a formal verification may easily become prohibitive. Therefore, realistic examples are necessary not only to evaluate the practical usefulness of a calculus but also to suggest adaptations and modifications that may increase its range of application. Unfortunately, there are only few such examples in the literature [Ossefort, 1983].

The goal of this chapter is to present the verification of a parallel program with a rather complex dynamical behavior whose correctness is not immediately clear. We do not start the verification with a nice formal model but first present the "real-world" problem that we actually want to be solved, then show how this problem can be modelled in a formal framework and finally derive the properties of this model.

The rest of this chapter is organized as follows:

- Section 3.2 describes the problem that we want to solve i.e. the verification of the implementation of the non-deterministic construct `pacWait`.
- Section 3.3 gives an introduction to temporal logic in the formulation of [Kröger, 1987]. We find this formulation too weak and introduce some extensions that make them more suitable for our purpose.
- Section 3.4 discusses the modelling of the program in the formal framework, a process that turns out at least as challenging and instructive as the verification itself.
- Section 3.5 describes the main lines of the verification based on some key lemmas; these lemmas and their proof are given in Appendix A.
- Section 3.6 describes the conclusions we have drawn from this case study and how they affect our view of parallel programming.

3.2 The Verification Goal

Our goal is to formally justify our confidence in the correctness of the implementation of the PACLIB result delivery protocol that has been presented in Section 2.4. We want to verify that this non-deterministic scheme is correct for *any* program that applies `pacStart` and `pacWait` under *all* circumstances (i.e. possible interleavings of operations).

For this purpose, we present the fraction of the source code that implements this scheme (the virtual task management is excluded):

- `pacShell` (Figure 3.1)is the function that is executed by any PACLIB task. It runs the denoted user function with the corresponding arguments and then calls `pacResult`.
- `pacResult` (Figure 3.3) delivers the result to all blocked tasks and finally stores the value into the result descriptor. The function uses several macros that perform the basic functions on result descriptors (e.g. `resLock`) and task descriptors (e.g. `taskLock`). The function `uWakeTask` awakes a blocked task.
- `pacWaitList` (Figure 3.4) is the basic function on which `pacWait` is based on. All task handles (i.e. result descriptors) are linked into a single list `rlist`. The function `uBlockTask` blocks the execution of the current task.

```
void pacShell(pacFunction f,
  pacObject a1, pacObject a2, pacObject a3, pacObject a4,
  pacObject a5, pacObject a6, pacObject a7, pacObject a8) {

  pacResult(f(a1, a2, a3, a4, a5, a6, a7, a8));
  pacDie();

}
```

Figure 3.1: `pacShell`

```
void pacResult(pacObject v) {

  pacObject value, value0;            /* function results */
  pacTask result, result0;            /* result descriptor */
  pacTaskList rlist;                  /* list of result descriptors */
  pacTaskList tlist, tlist0, tlist1;  /* lists of task descriptors*/
  uTask task, task0;                  /* task descriptors */
  int deliver;                        /* deliver flag */

  ...                                 /* code */
}

pacObject pacWaitList(pacTask *tp, pacTaskList rlist) {

  pacTaskList rlist0;                 /* temporary result descriptor list */
  pacTask result;                     /* actual result descriptor */
  uTask task;                         /* actual usystem task */
  pacObject value, v;                 /* actual result values */
  pacTaskList tlist, tlist0;          /* lists of waiting tasks */
  pacObject new;                      /* new list cell */
  int found = FALSE;                  /* value was found */

  ...                                 /* code */
}
```

Figure 3.2: Declaration of Local Variables

The given source code given is a slightly beautified, condensed and restructured version of the actual program. The actual implementation is more fault-tolerant, more optimized and more general than the functions shown there. The explanation of these details goes beyond the scope of the presentation and is of minor importance for our goal. However, please note that already at this level of discussion we have performed some simplifications.

3.3 The Temporal Logic of Programs

The formal framework we will use for reasoning is that of *temporal logic* as described in [Kröger, 1987]. The main reasons for this choice are the following:

- **Axiomatic Approach:** Temporal logic is an axiomatic approach to the formal treatment of parallel programs. It does not reason about the semantics of a program but about its properties. This is of particular importance since we want to deal with a whole class of (incompletely specified) programs.

```
void pacResult(pacObject v) {
  ...                                 /* declarations of local variables */

  result = uWorkTask->pac.result;     /* result descriptor of current task */

  resLock(result);                    /* lock descriptor */
  resGet(result, &task0, &value, &tlist); /* get associated task list */
  while (!listNull(tlist)) {          /* there are some waiting tasks */
    task = (uTask)listHead(tlist);    /* get next waiting task */
    resUnlock(result);                /* unlock result descriptor */

    taskLock(task);                   /* lock task descriptor */

    resLock(result);                  /* lock descriptor again */
    resGet(result, &task0, &value, &tlist);
    deliver = (!listNull(tlist)) && ((uTask)listHead(tlist) == task);
    resUnlock(result);                /* unlock descriptor again */

    if (deliver) {                    /* remove task from result descr */

      rlist = task->pac.wait;         /* results the task waited for */
      while (!listNull(rlist)) {      /* remove task from result descs */
        result0 = listHead(rlist);    /* get next result descriptor */
        resLock(result0);             /* lock result descriptor */
        resGet(result0, &task0, &value0, &tlist0);
        tlist1 = listNew();           /* empty predecessor list */
        while (task != (uTask)listHead(tlist0)) { /* search for task */
          tlist1 = tlist0;            /* new predecessor */
          tlist0 = listTail(tlist0);  /* task descriptor not yet found */
        }
        if (listNull(tlist1))         /* head of queue must be updated */
          resSetWait(result0, listTail(tlist0));
        else                          /* unlink the task from the queue */
          listSetTail(tlist1, listTail(tlist0));
        if (value0 == tlist0)         /* last task in list removed */
          resSetValue(result0, tlist1);
        resUnlock(result0);           /* unlock result descriptor */
        rlist = listTail(rlist);      /* get next descriptor */
      }

      task->pac.wait = result;        /* deliver result descriptor */
      task->pac.value = v;            /* deliver actual result value */
      uWakeTask(task);                /* wake up task */

    }

    taskUnlock(task);                 /* unlock this task */

    resLock(result);                  /* lock result descriptor */
    resGet(result, &task0, &value, &tlist); /* get associated task list */
  }

  resSetTask(result, AVAILABLE);      /* no task is executing any more */
  resSetValue(result, v);             /* this is the result value */
  resSetWait(result, NIL);            /* no tasks waiting any more */
  resUnlock(result);                  /* unlock result descriptor */

}
```

Figure 3.3: pacResult

```
pacObject pacWaitList(pacTask *tp, pacTaskList rlist) {
  ...                                /* declarations of local variables */

  taskLock(uWorkTask);               /* lock this task */
  rlist0 = rlist;                    /* list of result descriptors */

  while (!listNull(rlist0) && !found) { /* still more tasks to wait for */
    result = listHead(rlist0);       /* next result descriptor */
    resLock(result);                 /* lock descriptor and get values */
    resGet(result, &task, &value, &tlist);

    if (!taskAvail(task)) {          /* result not available */
      new = listCons((pacObject)uWorkTask, listNew());
      resSetValue(result, new);      /* tail of waiting queue */
      if (listNull(value))           /* no waiting tasks yet */
        resSetWait(result, new);     /* head of waiting queue */
      else                           /* there are already waiting tasks */
        listSetTail(value, new);     /* update tail of waiting queue */
      resUnlock(result);             /* unlock result descriptor */
      rlist0 = listTail(rlist0);     /* get rest of result descriptor list */
      continue;                      /* continue next loop iteration */
    }

    resUnlock(result);               /* unlock descriptor */
    if (tp) *tp = result;            /* remember result descriptor */
    v = value;                       /* result value */
    found = TRUE;                    /* value was found */

    while(rlist != rlist0) {         /* remove task from descriptors again */
      result = listHead(rlist);      /* next result descriptor */
      resLock(result);               /* lock descriptor */
      resGet(result, &task, &value, &tlist);
      tlist0 = listNew();            /* predecessor in list */
      while ((uTask)listHead(tlist) != uWorkTask) {
        tlist0 = tlist;              /* new predecessor */
        tlist = listTail(tlist);     /* next list tail */
      }
      if (listNull(tlist0))          /* no predecessor in list */
        resSetWait(result, listTail(tlist));
      else                           /* predecessor in list */
        listSetTail(tlist0, listTail(tlist));
      if (value == tlist) resSetValue(result, tlist0);
      resUnlock(result);             /* unlock result descriptor */
      rlist = listTail(rlist);       /* rest of list */
    }
  }

  if (found)                         /* value was found */
    taskUnlock(uWorkTask);           /* unlock current task */
  else {                             /* value not found */
    uWorkTask->pac.wait = rlist;     /* list of results that we wait for */
    taskUnlock(uWorkTask);           /* unlock current task */
    uBlockTask();                    /* block current task */
    if (tp) *tp = uWorkTask->pac.wait; /* this is the result descriptor */
    v = uWorkTask->pac.value;        /* this is the actual result */
    uWorkTask->pac.wait = listNew();   /* reset wait field */
    uWorkTask->pac.value = listNew();  /* reset value field */
  }
  return(v);                         /* return result */
}
```

Figure 3.4: `pacWaitList`

- **Intuitive Reasoning:** Temporal logic allows *intuitive* reasoning about programs since it just models our usual knowledge about time relationships. Details that are "obvious" for a human being can be skipped if it would be tedious to derive them from the basic axioms.
- **Hoare's Calculus:** As a special case, we can incorporate Hoare's calculus for the verification of sequential programs into the temporal logic framework. This allows to reason about sequential program parts in the common way.

This section gives the essential core of [Kröger, 1987]. However, we describe the material in a language that is more amenable to computer scientists. Moreover, in Subsection 3.3.6 we introduce some concepts that make the reasoning process more practicable.

3.3.1 Temporal Logic

The main idea of temporal logic is to incorporate the notion of *time* into the framework of first-order predicate logic. We define time to be an infinite, discrete and linearly ordered sequence of time points with a smallest element:

$$\tau_0 < \tau_1 < \tau_2 < \tau_3 < \ldots$$

The state of a universe $\mathcal{S}$ at each moment τ is defined by a valuation function η_τ that maps time-dependent quantities to values. Furthermore, a valuation function ξ maps time-invariant quantities to their denotations. We may thus define a *temporal structure* $\mathcal{K} = (\mathcal{S}, \xi, \eta)$ and a valuation function $\mathcal{K}_\tau$ that describes the semantics of a formula A at a given time point t. Please note that in general $\mathcal{K}_\tau(A) \neq \mathcal{K}_{\tau'}(A)$ for $\tau \neq \tau'$.

We define the semantics of atomic formulas and of the logical connectives as follows:

$$\begin{array}{lll}
\mathcal{K}_\tau(A) = \mathcal{S}^{(\xi,\eta_\tau)} & \multicolumn{2}{l}{\text{for every atomic formula } A} \\
\mathcal{K}_\tau(\neg A) = \mathrm{t} & \text{iff} & \mathcal{K}_\tau(A) = \mathrm{f} \\
\mathcal{K}_\tau(A \wedge B) = \mathrm{t} & \text{iff} & \mathcal{K}_\tau(A) = \mathrm{t} \text{ and } \mathcal{K}_\tau(b) = \mathrm{t} \\
\mathcal{K}_\tau(A \vee B) = \mathrm{t} & \text{iff} & \mathcal{K}_\tau(A) = \mathrm{t} \text{ or } \mathcal{K}_\tau(b) = \mathrm{t} \\
\mathcal{K}_\tau(A \rightarrow B) = \mathrm{t} & \text{iff} & \mathcal{K}_\tau(A) = \mathrm{f} \text{ or } \mathcal{K}_\tau(b) = \mathrm{t} \\
\mathcal{K}_\tau(A \leftrightarrow B) = \mathrm{t} & \text{iff} & \mathcal{K}_\tau(A) = \mathcal{K}_\tau(B)
\end{array}$$

For reasoning about time, we additionally introduce the unary *temporal operators* $\Box$ ("always") $\Diamond$ ("sometime") and $\bigcirc$ ("next time") and extend the valuation function $\mathcal{K}_\tau$ by

$$\begin{array}{lll}
\mathcal{K}_\tau(\Box A) = \mathrm{t} & \text{iff} & \mathcal{K}_{\tau'}(A) = \mathrm{t} \text{ for every } \tau' \geq \tau \\
\mathcal{K}_\tau(\Diamond A) = \mathrm{t} & \text{iff} & \mathcal{K}_{\tau'}(A) = \mathrm{t} \text{ for some } \tau' \geq \tau \\
\mathcal{K}_\tau(\bigcirc A) = \mathrm{t} & \text{iff} & \mathcal{K}_{\tau+1}(A) = \mathrm{t}
\end{array}$$

In other words, for a fixed moment τ, $\Box A$ is true, iff A will hold from τ on forever. The semantics of the other operators is then also immediately clear.

The quantifiers $\forall$ and $\exists$ are defined by

$$\begin{array}{lll}
\mathcal{K}_\tau(\forall x(A)) = \mathrm{t} & \text{iff} & \mathcal{K}'_\tau(A) = \mathrm{t} \text{ for every } \mathcal{K}' = (S, \xi', \eta') \\
 & & \text{with } \xi'(y) \stackrel{\mathcal{S}}{=} \xi(y) \text{ for every } y \text{ other than } x \\
\mathcal{K}_\tau(\exists x(A)) = \mathrm{t} & \text{iff} & \mathcal{K}'_\tau(A) = \mathrm{t} \text{ for some } \mathcal{K}' = (S, \xi', \eta') \\
 & & \text{with } \xi'(y) \stackrel{\mathcal{S}}{=} \xi(y) \text{ for every } y \text{ other than } x
\end{array}$$

We we also introduce the binary temporal operators **atnext**, **before** and **until** by

$$
\begin{array}{lll}
\mathcal{K}_\tau(A \texttt{ atnext } B) = \mathrm{t} & \text{iff} & \mathcal{K}_{\tau'}(A) = \mathrm{f} \text{ for every } \tau' \geq \tau \text{ or} \\
 & & \mathcal{K}_\tau(A) = \mathrm{t} \text{ for the smallest } \tau' > \tau \text{ with } \mathcal{K}'_\tau(B) = \mathrm{t} \\
\mathcal{K}_\tau(A \texttt{ before } B) = \mathrm{t} & \text{iff} & \text{if } \mathcal{K}_{\tau'} = \mathrm{t} \text{ for some } \tau' > \tau \\
 & & \text{then } \mathcal{K}_{\tau''} = \mathrm{t} \text{ for some } \tau'' \text{ with } \tau' > \tau'' > \tau \\
\mathcal{K}_\tau(A \texttt{ until } B) = \mathrm{t} & \text{iff} & \mathcal{K}_{\tau'} = \mathrm{t} \text{ for some } \tau' > \tau \text{ and} \\
 & & \mathcal{K}_{\tau''} = \mathrm{t} \text{ for every } \tau'' \text{ with } \tau' > \tau'' > \tau
\end{array}
$$

Similar to classical first-order predicate logic, may introduce the notion of validity into temporal logic.

Definition 1 (Validity) *A formula A is called valid in a temporal structure $\mathcal{K}$ ($\mathcal{K}$-valid) if $\mathcal{K}_\tau(A) = \mathrm{t}$ for every time τ. A is called valid if A is valid in every $\mathcal{K}$.*

Please note that the temporal notion of validity has one more "degree of freedom" than the classical notion: A is valid in the temporal sense only if it is true at every time point!

Similar to the transformation rules for logical connectives and quantifiers, there are many "temporal rules" like

$$
\begin{array}{lll}
\neg\Box A & \leftrightarrow & \Diamond\neg A \\
\bigcirc(A \rightarrow B) & \leftrightarrow & \bigcirc A \rightarrow \bigcirc B \\
\Box(A \rightarrow B) & \rightarrow & \Box A \rightarrow \Box B \\
\Box(A \wedge B) & \rightarrow & \Box A \wedge \Box B \\
\Diamond(A \vee B) & \rightarrow & \Diamond A \vee \Diamond B \\
\Box A \vee \Box B & \rightarrow & \Box(A \vee B) \\
\forall x(\bigcirc A) & \leftrightarrow & \bigcirc\forall x(A) \\
\exists x(\bigcirc A) & \leftrightarrow & \bigcirc\exists x(A) \\
\forall x(\Box A) & \leftrightarrow & \Box\forall x(A) \\
\exists x(\Diamond A) & \leftrightarrow & \Diamond\exists x(A) \\
A \texttt{ before } B & \leftrightarrow & (\neg B) \texttt{ atnext } (A \vee B)
\end{array}
$$

These rules are intuitively clear but can be easily derived from the definitions above e.g.

$$
\begin{array}{lll}
\mathcal{K}_\tau(\neg\Box A) = \mathrm{t} & \text{iff} & \text{not } \mathcal{K}_{\tau'}(A) = \mathrm{t} \text{ for every } \tau' \geq \tau \\
 & \text{iff} & \mathcal{K}_u(A) = \mathrm{f} \text{ for some } \tau' \geq \tau \\
 & \text{iff} & \mathcal{K}_u(\neg A) = \mathrm{t} \text{ for some } \tau' \geq \tau \\
 & \text{iff} & \mathcal{K}_\tau(\Diamond\neg A) = \mathrm{t}
\end{array}
$$

For a much more comprehensive list of temporal rules, see [Kröger, 1987].

3.3.2 Parallel Programs

A parallel program Π consists of a set processes $\Pi_1 \ldots \Pi_p$ each of which is a sequential program composed from the basic "assignment" statement and the statement builders "sequence" (the ; operator), "conditional" (one and two branch if statement) and "iteration" (while loop). All statements operate on the same global set of "program variables". For the synchronization of parallel programs, there is a special statement

await C **then** $v := E$

where C is a boolean condition and E is an expression to be assigned to the variable v. The intuitive semantics of this construct is that $v := E$ can be executed only if C is true. Otherwise the process Π_i executing the **await** statement is blocked until C becomes true.

A program has the form

initial R;
cobegin $\Pi_1 \parallel \ldots \parallel \Pi_p$ **coend**

where R is a first-order formula called the *initialization condition* that describes the initial state of the program. As an example, take the following program

initial $n = 0 \wedge sem$
cobegin
(α_0) **while** $i > 0$ **do**
(α_1) **await** sem **then** $sem :=$ false
(α_2) $stack[n] := i$
(α_2) $n := n + 1$
(α_3) $sem :=$ true
(α_4) $i := i - 1$
endwhile
(α_e) **stop**
$\parallel$
(β_0) **while** $j > 0$ **do**
(β_1) **await** sem **then** $sem :=$ false
(β_2) $stack[n] := j$
(β_2) $n := n + 1$
(β_3) $sem :=$ true
(β_4) $j := j - 1$
endwhile
(β_e) **stop**
coend

where two processes concurrently push items on a stack. A boolean variable sem (a "semaphore") is used for providing mutual exclusion of both processes during the stack access. Further on, we will use for semaphore operations the notation

$\mathbf{P}(sem) \equiv$ **await** sem **then** $sem :=$ false
$\mathbf{V}(sem) \equiv sem :=$ true

which is more common in parallel programming than the **await** construct. The **stop** statement in the program above is actually a shortcut for

stop $\equiv$ **await** false

and always denotes the last statement in each parallel process beyond which execution cannot proceed. All statements in the program are labelled for a unique reference. The set of all labels in a program Π is denoted by $\mathcal{M}_\Pi$; the set of all labels without the trailing **stop** labels is denoted by $\overline{\mathcal{M}_\Pi}$.

3.3.3 Program Execution

The computational behavior of a parallel program is described by an *interleaving model* of execution where at each time point one atomic operation of a single process is executed.

Definition 2 (Atomic Operation) *An atomic operation is*

- *an assignment statement,*
- *an* **await** *statement (test of condition and execution of the assignment),*

- *a test of the condition in an* **if** *statement (and entry in the respective branch),*
- *a test of the condition in an* **while** *statement (and entry into or exit from loop).*

Definition 3 (Program State) *The state of a program* Π *at a time point* τ *is a tuple*

$$\eta_\tau = (\mu, \lambda_1, \ldots, \lambda_p, \kappa)$$

where

- $\mu(a)$ *is the value of the program variable* a *at time* τ,
- λ_i *is the label of the statement of* Π_i *that is ready to be executed at time* τ,
- $\kappa \neq 0$ *means that* Π_κ *is the process (and thus* λ_κ *is the statement) scheduled for execution at time* τ,
- $\kappa = 0$ *means that no* Π_i *may be scheduled at* τ *and therefore also not at any time* $\tau' > \tau$ *(deadlock or terminal state).*

The possible transitions from one program state to another will be described by a set of tuples (λ, C, λ') saying that, if the condition C is true, execution may proceed from statement λ to λ'.

Definition 4 (Transitions) *The set* $trans(\Pi)$ *of possible transitions of a sequential program* Π *is inductively defined on the structure of* Π :

1. **Stop:** If Π is the statement

   $\lambda_l :$ **stop**

   then $trans(\Pi) = \emptyset$.
2. **Assignment:** If Π is an assignment

   $\lambda_l : \ v := E$
   $\lambda_m : \ \ldots$

   then $trans(\Pi) = \{(\lambda_l, \mathrm{t}, \lambda_m)\}$.
3. **Wait Statement:** If Π is a wait statement

   $\lambda_l :$ **await** C **then** ...
   $\lambda_m : \ \ldots$

   then $trans(\Pi) = \{(\lambda_l, C, \lambda_m)\}$.
4. **Sequence:** If Π is a sequence

   $\lambda_l : \ \Pi_1$
   $\lambda_m : \ \Pi_2$
   $\lambda_n : \ \ldots$

   then $trans(\Pi) = trans(\Pi_1) \cup trans(\Pi_1)$.
5. **Two Branch Conditional:** If Π is a two branch conditional

   $\lambda_l :$ **if** C **then** $\lambda_m : \Pi_1$ **else** $\lambda_n : \Pi_2$ **endif**
   $\lambda_o : \ \ldots$

   then $trans(\Pi) = \{(\lambda_l, C, \lambda_m), (\lambda_l, \neg C, \lambda_m)\} \cup trans(\Pi_1) \cup trans(\Pi_2)$.
6. **One Branch Conditional:** If Π is a one branch conditional

   $\lambda_l :$ **if** C **then** $\lambda_m : \Pi_1$ **endif**
   $\lambda_n : \ \ldots$

   then $trans(\Pi) = \{(\lambda_l, C, \lambda_m), (\lambda_l, \neg C, \lambda_n)\} \cup trans(\Pi_1)$.

7. **While Loop** If Π is a while loop

$$\begin{array}{l}\lambda_l : \textbf{ while } C \textbf{ do } \lambda_m : \Pi_1 \textbf{ endwhile} \\ \lambda_n : \ \ldots\end{array}$$

then $trans(\Pi) = \{(\lambda_l, C, \lambda_m), (\lambda_l, \neg C, \lambda_n)\} \cup trans(\Pi_1)$.

In this informal definition, the trailing statement "..." denotes the successor of the current statement (such a successor exists and is unique for all statements but the final **stop** statement in a process). The exact definition of trans(Π) can be found in [Kröger, 1987].

Definition 5 (Statement Execution) *For every $\alpha \in \mathcal{M}_\Pi$, we define the execution condition*

$$E_\alpha = \begin{cases} C, & \text{if } \alpha : \texttt{ await } C \texttt{ then } v := E \\ \text{true}, & \text{otherwise} \end{cases}$$

We may now define the structure of valid execution sequences of a program.

Definition 6 (Execution Sequence) *An execution sequence of Π is an infinite sequence $\eta = (\eta_0, \eta_1, \eta_2, \ldots)$ of program states such that*

1. **Initial State:** $\eta_0 = (\mu, \alpha_0^1, \ldots, \alpha_0^p, \kappa)$ *and* $\mathcal{K}_0(R) = \mathrm{t}$.
2. **Transition:** *If* $\eta_\tau = (\mu, \lambda_1 \ldots \lambda_\kappa \ldots \lambda_p, \kappa > 0)$ *then* $\eta_{t+1} = (\mu', \lambda_1 \ldots \lambda'_\kappa \ldots \lambda_p, \kappa')$ *and, for some formula C,* $(\lambda_\kappa, C, \lambda'_\kappa) \in trans(\Pi_\kappa)$ *and* $\mathcal{K}_\tau(C) = \mathrm{t}$.
3. **Terminal State:** *If* $\eta_\tau = (\mu, \lambda_1, \ldots, \lambda_p, 0)$ *then* $\eta_{\tau+1} = \eta_\tau$ *and* $\mathcal{K}_\tau(E_{\lambda_i}) = \mathrm{f}$ *for every* $1 \leq i \leq p$.

Please note that this definition does not give the semantics of the assignment and `await` statement since it does not describe the relationship between μ and μ' on a state transition.

3.3.4 Temporal Program Semantics

The language of temporal logic described in Subsection 3.3.1 is not yet sufficient to describe *program states* since it does not give any possibility to express information about the program counters $\lambda_1, \ldots, \lambda_p, \kappa$ in a program. For this purpose, we introduce the following set of propositional variables:

1. $\texttt{at}\{\lambda\}$ for every $\lambda \in \mathcal{M}_\Pi$.
2. λ for every $\lambda \in \overline{\mathcal{M}}_\Pi$.

The informal meaning of the proposition $\texttt{at}\{\lambda\}$ is "the statement λ is ready for execution in some process Π_i" while λ means "the statement λ is executed next". The information expressed by the program state

$$\eta_\tau = (\mu, \lambda_1, \ldots, \lambda_p, \kappa)$$

with $\kappa > 0$ can then be expressed by the formula

$$(v_1 = \mu(v_1)) \wedge \ldots \wedge (v_n = \mu(v_n)) \wedge \texttt{at}\{\lambda_1\} \wedge \ldots \wedge \texttt{at}\{\lambda_p\} \wedge \lambda_\kappa$$

where the v_i are the program variables occurring in Π.

In the notion of temporal logic, we may therefore interpret a program state η_τ as a valuation function

$$\begin{array}{lll} \eta_\tau(v) = \mu(v) & & \text{for every program variable v} \\ \eta_\tau(\lambda) = \mathrm{t} & \text{iff} & \lambda = \lambda_\kappa \text{ and } \kappa \neq 0 \\ \eta_\tau(\texttt{at}\{\lambda\}) = \mathrm{t} & \text{iff} & \lambda = \lambda_j \text{ for some } j \end{array}$$

We may now apply the notion of validity defined in Subsection 3.3.1 to the special case of execution sequences.

Definition 7 (Π-Validity) *A formula A is Π-valid if it is true for every execution sequence η of Π (at any time τ).*

However, the restriction of temporal structures to execution sequences of programs can be also described by giving additional axioms and rules for deriving more formulas. We will distinguish between two kinds of axioms

1. Structural Axioms that are invariant for any program Π,
2. Computation Scheme Axioms that depend on Π.

[Kröger, 1987] proves that a formula is Π-valid if it can be derived using these axioms where the following abbreviation is used:

Definition 8 (Initial and Terminal State) *For any process Π with $\mathcal{M}_\Pi = \{\alpha_0, \ldots, \alpha_l\}$ we define*

$$\begin{array}{lcl} \mathrm{start}_\Pi & \equiv & \mathtt{at}\{\alpha_0^i\} \wedge \ldots \wedge \mathtt{at}\{\alpha_0^p\} \wedge R \\ \mathrm{nil}_\Pi & \equiv & \neg\alpha_1 \wedge \neg\alpha_2 \wedge \ldots \neg\alpha_l \end{array}$$

The informal meaning of start_Π is "Execution of Π is in its initial state" while nil_Π says "No statement is executed in Π". The structural axioms are then the following:

Rule 1 (Basic Rule) *A is always true if it holds permanently beginning with the initial state of the execution sequence:*

$$\mathrm{start}_\Pi \rightarrow \Box A \vdash A.$$

Axiom 1 (Basic Axiom) *If no statement is executed, nothing changes:*

$$\mathrm{nil}_\Pi \wedge A \rightarrow \bigcirc(\mathrm{nil}_\Pi \wedge A).$$

Axiom 2 (Two Execute) *No two statements execute at the same time:*

$$\forall\lambda \neq \lambda'(\lambda \rightarrow \neg\lambda').$$

Axiom 3 (Execute) *An statement may only execute if it is ready to:*

$$\forall\lambda(\lambda \rightarrow \mathtt{at}\{\lambda\}).$$

Axiom 4 (Two Ready) *In every Π_k, no two statements are ready to execute at the same time:*

$$\forall k, l, m(\mathtt{at}\{\lambda_l^k\} \wedge l \neq m \rightarrow \neg\,\mathtt{at}\{\lambda_m^k\}).$$

Axiom 5 (Enabled) *If some statement is enabled, then some statement has to execute:*

$$\forall\lambda(\mathtt{at}\{\lambda\} \wedge E_\lambda \rightarrow \neg\mathrm{nil}_\Pi).$$

Axiom 6 (Ready) *An statement ready to execute but not executing remains ready:*

$$\forall\lambda(\mathtt{at}\{\lambda\} \wedge \neg\lambda \rightarrow \bigcirc\,\mathtt{at}\{\lambda\}).$$

Axiom 7 (End) *If Π_i has reached its end label then it remains there:*

$$\forall k(\mathtt{at}\{\lambda\}_e^k \rightarrow \bigcirc\,\mathtt{at}\{\lambda\}_e^k).$$

Axiom 8 (Change) *Every non-assignment and non-wait statement does not cause any variable changes:*

$$\forall\lambda \notin \{:=, \mathit{await}\}\ \forall P \notin \mathcal{L}_\mathrm{P}(\lambda \wedge P \rightarrow \bigcirc P).$$

The set of axioms describing the behavior of a particular program Π can be derived from the following axiom:

Axiom 9 (Computation Scheme) *For every $\lambda \in \mathcal{M}_\Pi$, the following holds*

$$\lambda \rightarrow (C_1 \wedge \bigcirc\, \mathtt{at}\{\lambda_1\}) \vee \ldots \vee (C_q \wedge \bigcirc\, \mathtt{at}\{\lambda_q\}).$$

where $(\lambda, C_1, \lambda_1), \ldots, (\lambda, C_q, \lambda_q)$ are all transitions with first statement λ in trans(Π).

Actually, this axiom is more general than necessary and can (according to the definition of trans(Π)) be replaced by the following set of axioms that are more intuitive:

Axiom 10 (Assignment Statement) *If λ_l is an assignment statement i.e.*

$\lambda_l :\ v := E$
$\lambda_m :\ \ldots$

then execution proceeds from λ_l to λ_m:

$$\lambda_l \rightarrow \bigcirc\, \mathtt{at}\{\lambda_m\}.$$

Axiom 11 (Wait Statement) *If λ_l is a wait statement i.e.*

$\lambda_l :$ **await** C **then**...
$\lambda_m :\ldots$

then the condition C lets execution either suspend or proceed from λ_l to λ_m:

$$\lambda_l \rightarrow (C \rightarrow \bigcirc\, \mathtt{at}\{\lambda_m\}) \wedge (\neg C \rightarrow \bigcirc\, \mathtt{at}\{\lambda_l\}).$$

Axiom 12 (Two Branch Conditional) *If λ_l is the label of a two branch conditional i.e.*

$\lambda_l :$ **if** C **then** $\lambda_m :\ \ldots$ **else** $\lambda_n :\ \ldots$**endif**

then the condition C lets execution proceed from λ_l to one of the branches λ_m or λ_n, respectively:

$$\lambda_l \rightarrow (C \rightarrow \bigcirc\, \mathtt{at}\{\lambda_m\}) \wedge (\neg C \rightarrow \bigcirc\, \mathtt{at}\{\lambda_n\}).$$

Axiom 13 (One Branch Conditional) *If λ_l is the label of a one branch conditional i.e.*

$\lambda_l :$ **if** C **then** $\lambda_m :\ \ldots$ **endif**
$\lambda_n :\ \ldots$

then the condition C lets execution proceed from λ_l either to the branch λ_m or to the successor λ_n:

$$\lambda_l \rightarrow (C \rightarrow \bigcirc\, \mathtt{at}\{\lambda_m\}) \wedge (\neg C \rightarrow \bigcirc\, \mathtt{at}\{\lambda_n\}).$$

Axiom 14 (While Loop) *If λ_l is the label of a while loop i.e.*

$\lambda_l :$ **while** C **do** $\lambda_m :\ldots$ **endwhile**
$\lambda_n :\ldots$

then the condition C lets execution either proceed to the next iteration of the body λ_m or to the successor λ_n:

$$\lambda_l \rightarrow (C \rightarrow \bigcirc\, \mathtt{at}\{\lambda_m\}) \wedge (\neg C \rightarrow \bigcirc\, \mathtt{at}\{\lambda_l\}).$$

The effect of the assignment and the `await` statement is described by the following

Axiom 15 (Assignment) *For every statement*

$\lambda: \ v := E$

and every statement

$\lambda:$ `await` C `then` $v := E$

the following holds for every non-temporal formula P:

$\lambda \wedge P_v(E) \rightarrow \bigcirc P.$

The whole set of axioms describing the properties of a program Π is also called the *temporal semantics* of Π.

3.3.5 Description of Program Properties

There are basically two kinds of program properties that are of interest:

1. **Safety (or Invariance) Properties**,
2. **Liveness (or Eventuality) Properties**.

Invariance properties are expressed by formulas of the form

$A \rightarrow \Box B.$

For instance, the *partial correctness* of a program with precondition P and postcondition Q can be expressed by the formula

$\mathrm{start}_\Pi \wedge P \rightarrow \Box((\ \mathtt{at}\{\alpha_e^1\} \wedge \ldots \wedge \mathtt{at}\{\alpha_e^p\}) \rightarrow Q)$

saying that if Π starts with P then Q always holds in the terminal state.

Eventuality properties are expressed by formulas of the form

$A \rightarrow \Diamond B.$

As an example take the *termination* of a program with precondition P that can be written

$\mathrm{start}_\Pi \wedge P \rightarrow \Diamond(\ \mathtt{at}\{\alpha_e^1\} \wedge \ldots \wedge \mathtt{at}\{\alpha_e^p\})$

saying that if Π starts with P then it will eventually reach the terminal state.

The proof of invariant properties can be based on the following rule

Rule 2 (Invariant) *In order to prove "if A now holds then B always holds from now on", it suffices to prove that B follows from A and that B is invariant under every statement:*

$(A \rightarrow B) \wedge \forall \lambda \in \overline{\mathcal{M}}_\Pi\ (\lambda \wedge B \rightarrow \bigcirc B)\ \vdash\ A \rightarrow \Box B.$

For proving eventuality properties, we are faced with the problem that some "natural" program properties can not be derived from the set of axioms specified up to now. Consider e.g. the program

initial $a = 1$
cobegin
(α_0) $a := 0$
(α_e) **stop**
||
(β_0) **while** $a = 1$ **do**
(β_1) $b := 0$
endwhile
(β_e) **stop**
coend

Clearly we would claim that in any reasonable model of parallel execution Π should terminate sometime i.e. α_e should be reached. In other words,

$$\text{start}_\Pi \rightarrow \Diamond\, \mathtt{at}\{\alpha_e\}$$

should be Π-valid but unfortunately it is not. The reason for this is that there is one execution sequence in which above formula is not true, namely that where forever β_0 and β_1 are alternately scheduled for execution. In order to overcome this deficiency, we introduce the notion of "fairness":

Definition 9 (Fairness) *An execution sequence $\eta = (\eta_0, \eta_1, \eta_2, \ldots)$ of a program Π is called fair it has the following property: If there are infinitely many states*

$$\eta_\tau = (\mu, \lambda_1, \ldots, \lambda_i, \ldots, \lambda_p, \kappa)$$

containing the same λ_i such that $\mathcal{K}_\tau(E_{\lambda_i}) = \mathrm{t}$, then $\kappa = i$ in infinitely many η_τ.

From now on, a formula A is considered Π-valid if it is valid in every *fair* execution sequence. Equivalently, this restriction can be described by the following

Axiom 16 (Fairness) *If an statement is enabled infinitely often then it will be executed sometime:*

$$\forall\lambda((\Box\Diamond(\,\mathtt{at}\{\lambda\} \wedge E_\lambda)) \rightarrow \Diamond\lambda).$$

This notion of fairness is also compatible with all process schedulers that apply a *round-robin* strategy with *preemptive execution* of processes (like the PACLIB scheduler).

Definition 10 (Process Execution) *For any process Π_i with $\overline{\mathcal{M}}_{\Pi_i} = \{\alpha_0^i, \ldots, \alpha_m^i\}$ we define the execution condition E^i as*

$$E^i \equiv (\,\mathtt{at}\{\alpha_0^i\} \wedge E_{\alpha_0^i}) \vee \ldots \vee (\,\mathtt{at}\{\alpha_0^e\} \wedge E_{\alpha_0^e})$$

The informal meaning of E^i is "one of the statements in Π^i is enabled for execution". This predicate is used in the following rule on which the proof of eventually properties is based.

Rule 3 (Well Founded Orderings) *To prove that B will eventually hold it suffices to prove that there is a well-founded ordering $(Z, \preceq)$ and a term z such that no statement in Π increases z with respect to this ordering, that there is for each value of z a ""helpful" process $\Pi_{h(z)}$ decreasing z and that $\Pi_{h(z)}$ will be eventually scheduled for execution:*

$$\begin{array}{ll} & \alpha \wedge A(z) \rightarrow \bigcirc(B \vee \exists z'(z' \preceq z \wedge A(z'))) \text{ for every } \alpha \in \overline{\mathcal{M}}_\Pi \backslash \overline{\mathcal{M}}_{\Pi_{h(z)}}, \\ & \alpha \wedge A(z) \rightarrow \bigcirc(B \vee \exists z'(z' \prec z \wedge A(z'))) \text{ for every } \alpha \in \overline{\mathcal{M}}_{\Pi_{h(z)}} \\ & \Box A(z) \rightarrow \Diamond(B \vee E^{\Pi_{h(z)}}) \\ \vdash & \exists z(A(z)) \rightarrow \Diamond B \end{array}$$

3.3.6 Time Constants

The standard framework of temporal logic does not give any possibility for explicitly referencing time points. [Kröger, 1987] says

> Consider now a proposition A. In order to describe the possible variety of the truth values of A at different times t the simplest linguistic means would be to introduce an explicit time parameter in the proposition and and denote it by $A(t)$. The main idea of the logical language we want to define, however, is to avoid just such an explicit occurrence of time. The reason is that we do not really want to be able to express assertions like "A is true at t_{17}". We rather want to have at hand logical *operators* which enable us to formulate new propositions about the truth values of A at time points which are related to some reference point (the *present time*) in particular ways.

We agree with the statement that we are not interested in expressing assertions like "A holds at t_{17}". However, we argue that time references are nevertheless useful for using

1. different "reference systems" in formulas,
2. a more general induction rule for proving.

The first item addresses the fact that even within a fixed temporal structure $\mathcal{K}$ the assertion

$$A$$

has not a unique meaning since it does not say *when* A holds. By just writing down A, it is not clear whether we want to say $\mathcal{K}_{17}(A) = \mathrm{t}$ or $\mathcal{K}_{23}(A) = \mathrm{t}$ or $\mathcal{K}_\tau(A) = \mathrm{t}$ where τ is the "present time". The usual solution is to consider A as a shortcut for

$$\Box A.$$

However, the definition of the $\Box$ operator

$$\mathcal{K}_\tau(\Box A) = \ \mathrm{t} \text{ if } \mathcal{K}_{\tau'}(\Box A) = \mathrm{t} \text{ for every } \tau' > \tau$$

shows that we have shifted the problem only to another level. The denotation of $\Box A$ still depends on the reference point τ considered as the "present time". In program verification, the usual attitude is to consider the the start time of a program as the reference time 0 and to express all assertions relative to this time. Therefore, A is actually a shortcut for

$$\mathcal{K}_0(\Box A) = \mathrm{t}.$$

If we want to reference another time point τ_A during a program run, we therefore have to find a unique characterization by a condition A on the state of the program variables and of the location counters such that

$$\mathcal{K}_{\tau_A}(A) = \mathrm{t} \text{ and } \mathcal{K}_\tau(A) = \ \mathrm{f} \text{ for every } \tau \neq \tau_A.$$

The assertion "B holds at τ_A" is then expressed by $A \rightarrow B$. However, this indirect way of referencing time points makes propositions often difficult to understand. We have to remember that the formula is a shortcut for $\mathcal{K}_0(\Box(A \rightarrow B)) = \mathrm{t}$ which can be reduced to

$$\mathcal{K}_{\tau_A}(B) = \mathrm{t}.$$

what we actually would like to say. The problem is that the one-to-one correspondence between A and τ_A is not a priori clear from the formula. This is especially annoying if we want to switch to another reference time i.e. to formulate a set of formulas relatively to some τ_A. We therefore propose to directly reference particular time points by the introduction of time constants.

Definition 11 (Time Constants) *For every time τ there is a constant τ defined by*

$$\begin{aligned} \mathcal{K}_{\tau'}(\tau) &= \mathrm{t} \quad \text{if } \tau = \tau' \\ &= \mathrm{f} \quad \text{if } \tau \neq \tau' \end{aligned}$$

A formula $\tau \rightarrow A$ is therefore a shortcut for $\mathcal{K}_0(\Box(\tau \rightarrow A))$ which can be reduced to

$$\begin{aligned} \mathcal{K}_0(\Box(\tau \rightarrow A)) = \ \mathrm{t} \quad & \text{if } \mathcal{K}_{\tau'}(\tau \rightarrow A) = \mathrm{t} \text{ for every } \tau' \geq 0 \\ & \text{if } \mathcal{K}_\tau(A) = \mathrm{t} \end{aligned}$$

i.e. the interpretation of the formula is directly clear from its syntactic structure. In a similar way, the interpretation of $\tau_0 \rightarrow (A \ \mathtt{until}\ \tau_1)$ can be reduced to

$$\mathcal{K}_0(\Box(\tau_0 \rightarrow (A \ \mathtt{until}\ \tau_1))) = \ \mathrm{t} \text{ if } \mathcal{K}_\tau(A) = \ \mathrm{t} \text{ for every } \tau \text{ with } \tau_0 \leq \tau < \tau_1$$

which is immediately clear from the formula itself.

The purpose of time constants is therefore to constrain the reference time (respectively the time range) of a formula and to make this constraint explicit in the formula. We will therefore say that τ is the reference time of A in a context $\tau \rightarrow A$ and that the interval $[\tau_0 \ldots \tau_1[$ is the time range of A in a context $\tau_0 \rightarrow (A \; \texttt{until} \; \tau_1)$. We will in our verification only use such constrained formulas.

Furthermore, we introduce an additional structural program axiom that defines a special time constant for the start time of a program Π:

Axiom 17 (Start) *τ_Π is the start time of the program Π:*

$$\tau_\Pi \rightarrow \mathrm{start}_\Pi.$$

This implies that in the context of program verification for each time τ the assertion $\tau > \tau_\Pi$ holds. By Axiom 8 (Initial State), we then know

$$\tau_\Pi \rightarrow \Box A \vdash A$$

The *definition* of a new time constant τ will always depend on previously defined time constants starting with τ_Π. This definition will be based on a characteristic condition A (as explained above) e.g. by saying

$$\tau_\Pi \rightarrow \tau \; \texttt{atnext} \; A$$

(where we have to prove that $\tau_\Pi \rightarrow \Diamond A$). After this definition, we may therefore use the symbol τ to reference the denoted time and apply the assertion $\tau \rightarrow A$.

A second advantage of introducing time constants is the possibility of applying a generalized induction rule. A proof guided by Rule 2 (Invariant) only allows conclusions from one statement to the next. We think that this view does not reflect the way most human beings intuitively reason about a program. Consider e.g. the program

initial $n = 0 \wedge N \geq 0 \wedge sem$
cobegin $\Pi_1 \parallel \ldots \parallel \Pi_p$ **coend**

where each Π_i is constructed as follows:

(α_0^i)	**P**(sem)
(α_1^i)	**while** $n < N$ **do**
(α_2^i)	$\quad s_i := n$
(α_3^i)	$\quad n := n + 1$
(α_4^i)	$\quad$ **V**(sem)
(α_5^i)	$\quad a[s_i] := s_i$
(α_6^i)	$\quad$ **P**(sem)
	endwhile
(α_7^i)	**V**(sem)
(α_e^i)	**stop**

Suppose that we want to verify

(3.1) $\Box(\forall i(\texttt{at}\{\alpha_e^i\}) \rightarrow \forall j < N(a[j] = j)).$

i.e. that at the termination of the program $a[j] = j$ holds for every j.

Proof: For proving this assertion we describe the temporal behavior of the program by the formula

$A \equiv$
$\quad (n \leq N,$

$$\begin{aligned}
& sem \rightarrow (B_n \wedge \forall i(C_i)),\\
& \neg sem \rightarrow \exists k(C_k \wedge \forall i \neq k(\neg C_k)),\\
& \forall i(\,\mathtt{at}\{\alpha_1^i\} \rightarrow \neg sem \wedge B_n),\\
& \forall i(\,\mathtt{at}\{\alpha_2^i\} \rightarrow \neg sem \wedge B_n \wedge n < N \wedge E),\\
& \forall i(\,\mathtt{at}\{\alpha_3^i\} \rightarrow \neg sem \wedge B_n \wedge n < N \wedge s_i = n \wedge E),\\
& \forall i(\,\mathtt{at}\{\alpha_4^i\} \rightarrow \neg sem \wedge B_{n-1} \wedge n \leq N \wedge s_i = n-1 \wedge E),\\
& \forall i(\,\mathtt{at}\{\alpha_5^i\} \rightarrow s_i < n),\\
& \forall i(\,\mathtt{at}\{\alpha_7^i\} \rightarrow \neg sem \wedge n = N \wedge B_n),\\
& \forall i(\,\mathtt{at}\{\alpha_e^i\} \rightarrow n = N \wedge B_n)).
\end{aligned}$$

where B_n, C_i and D are defined as

(3.2) $B_n \equiv \forall j < n(a[j] \neq j \rightarrow \exists k(\,\mathtt{at}\{\alpha_5^k\} \wedge s_k = j))$,

(3.3) $C_i \equiv \,\mathtt{at}\{\alpha_1^i, \ldots, \alpha_4^i, \alpha_7^i\}$,

(3.4) $D \equiv \forall k(\neg\,\mathtt{at}\{\alpha_e^k\})$.

The truth of (3.1) is a direct consequence of $\Box A$.

For proving $\Box A$, we have to prove by Rule 2 (Invariant) $\text{start}_\Pi \rightarrow A$ (which is easy) and, for each λ, $\lambda \wedge A \rightarrow \bigcirc A$. We will only show the case $\lambda = \alpha_4^i$ (for some i):

1. α_4^i and $n \leq N$ imply $\bigcirc n \leq N$.
2. α_4^i and A imply $\neg sem$, B_{n-1} and $s_i = n-1$. A and $\neg sem$ imply $\exists k(C_k \wedge \forall i \neq k(\neg C_k))$; α_4^i implies C_i and $\bigcirc\neg C_i$ and therefore $\bigcirc\forall i(\neg C_i)$. B_{n-1}, $s_i = n-1$ and $\bigcirc\,\mathtt{at}\{\alpha_5^i\}$ imply $\bigcirc B_n$.
3. α_4^i implies $\bigcirc sem$.
4. $\bigcirc\forall i(\neg C_i)$ and $\neg E$ imply $\bigcirc\neg\,\mathtt{at}\{\alpha_1^{i'}, \ldots, \alpha_4^{i'}, \alpha_7^{i'}, \alpha_e^{i'}\}$ for all i'. $\bigcirc\,\mathtt{at}\{\alpha_5^{i'}\}$ implies $\mathtt{at}\{\alpha_5^{i'}\}$, therefore $s_{i'} < n$ and thus $\bigcirc s_{i'} < n$. $\Box$

Please note that have only shown one of eight cases (for each α_l^i in the program)! A full verification of even this toy program becomes therefore very complex. This "statement-oriented" view is therefore not feasible for the verification of "real" parallel programs since the overview on the main lines of the proof easily goes lost. However, by the introduction of explicit time references we can use the following generalized induction rule that we think is more easier to use also for "semi-formal" proofs:

Rule 4 (Induction) *In order to prove "if A holds at τ_A then B always holds from τ_A on", it suffices to prove that B follows from A at τ_A and that B is true at every τ under the assumption that it has been true from τ_A until τ:*

$$\begin{aligned}
&(\tau_A \rightarrow (A \rightarrow B)) \wedge \forall \tau > \tau_A((\tau_A \rightarrow (B \;\mathtt{until}\; \tau)) \rightarrow (\tau \rightarrow B)) \;\vdash\\
&\quad \tau_A \rightarrow (A \rightarrow \Box B).
\end{aligned}$$

We will apply this rule to the program presented before. The main ideas there are

1. sem guarantees mutual exclusion on n,
2. B_n holds outside the critical region protected by sem,
3. $n = N$ holds at the end of the program.

Proof: The mutual exclusion property

(3.1) $\Box((sem \rightarrow \forall i(C_i)) \wedge (\neg sem \rightarrow \exists i(C_i \wedge \forall k \neq i(\neg C_k))))$

is easy to verify since α_0^i and α_6^i are the only entries and α_4^i and α_7^i the only exits from the critical region. Now let us prove that

(3.2) $\Box(sem \rightarrow B_n)$

By Rule 4 (Induction Rule), we have to prove $\tau_\Pi \rightarrow B_n$ which is clear since $\tau_\Pi \rightarrow n = 0$. Now take an arbitrary fixed time $\tau > \tau_\Pi$ and assume

(3.3) $\tau_\Pi \rightarrow ((sem \rightarrow B_n)\ \mathtt{until}\ \tau)$

(3.4) $\tau \rightarrow sem$

We have to prove $\tau \rightarrow B_n$. Let $\tau' = \tau - 1$. We only have to consider those cases where the truth value of $sem \rightarrow B_n$ may become false i.e. where $\tau' \rightarrow \neg sem$ (at the exits of critical regions) or where indeed $\tau' \rightarrow sem$ (and thus by the induction hypothesis $\tau' \rightarrow B_n$) but perhaps $\tau \rightarrow \neg B_n$. Actually, there are only the following cases are of interest:

1. $\boldsymbol{\tau' \rightarrow (\alpha_4^i \vee \alpha_7^i)}$: We know $\tau' \rightarrow \neg sem$. Let τ'' be the largest $\tau'' < \tau'$ with $\tau'' \rightarrow sem$. By the induction hypothesis, we then know $\tau'' \rightarrow B_n$. By the program structure, we know $\tau'' \rightarrow (\alpha_0^i \vee \alpha_6^i)$ and $\tau'' \rightarrow \bigcirc\alpha_1^i$. By (3.1), we know $\tau'' \rightarrow (s_i = n-1 \wedge B_{n-1})\ \mathtt{atnext}\ \alpha_4^i$ i.e. $\tau' \rightarrow (s_i = n-1 \wedge B_{n-1})$. We thus have $\tau \rightarrow B_n$.

2. $\boldsymbol{\tau' \rightarrow \alpha_5^i}$: If $\tau' \rightarrow sem$, we know by the induction hypothesis $\tau' \rightarrow B_n$. We also know $\tau' \rightarrow \mathtt{at}\{\alpha_5^i\}$ and $\tau' \rightarrow \bigcirc\neg\,\mathtt{at}\{\alpha_5^i\}$ but also $\tau' \rightarrow \bigcirc a[s_i] = s_i$. Thus $\tau' \rightarrow \bigcirc B_n$.

   If $\tau' \rightarrow \neg sem$, let τ'' be the largest $\tau'' < \tau'$ with $\tau'' \rightarrow sem$. By the induction hypothesis, we know $\tau'' \rightarrow B_n$. Since $\tau'' \rightarrow \neg sem\ \mathtt{until}\ \tau'$, it is by mutual exclusion and the program structure clear that, for any $\bar{n}$, $\tau'' \rightarrow n = \bar{n}$ implies $\tau' \rightarrow n \leq \bar{n} + 1$ and therefore $\tau' \rightarrow B_{n-1}$. This implies $\tau \rightarrow B_n$ with the above argument.

The fact that

(3.5) $\tau_\Pi \rightarrow \Box\forall i(\,\mathtt{at}\{\alpha_e^i\} \rightarrow n = N)$

can be checked as follows. Let τ and i be arbitrary fixed such that

(3.6) $\tau \rightarrow \mathtt{at}\{\alpha_e^i\}$.

By the program structure, it is clear that $\tau \rightarrow n \geq N$. Now assume $\tau \rightarrow n > N$. In this case, there exists some $\tau' < \tau$ and some i' such that

(3.7) $\tau' \rightarrow (n = N \wedge \alpha_3^{i'})$.

By (3.1), this implies that for the largest $\tau'' < \tau'$ with $\tau'' \rightarrow \alpha_2^{i'}$ we have $\tau'' \rightarrow n = N$ and therefore $\tau'' \rightarrow \bigcirc\,\mathtt{at}\{\alpha_7^{i'}\}$ which contradicts (3.7). $\Box$

The application of Rule 4 (Induction) does of course principally not save any work; the complexity of a formal proof is as large as a proof carried through by Rule 2 (Invariant) (since in the induction step a case distinction over all statements is required). The main advantage of this new rule is that it allows to "jump back" in one step to a past time-point where something "interesting" has happened (e.g. to the entry of a critical region).

We may then apply the induction hypothesis to this time point and proceed to the "current" time point. The reasoning on the "uninteresting" intermediate points (e.g. within a critical region) may be then less formal. In the "statement-oriented" style of reasoning, such jumps are not possible. There the behavior of the program has to be described for each statement and verification has to proceed from one statement to the next.

For the use of Rule 4 (Induction), we introduce the "backward" oriented time operators $\triangle$ ("previous time") and `atprev`:

$\mathcal{K}_\tau(\triangle A) = $ t	iff	$\mathcal{K}_{\tau-1}(A) = $ t and $\tau > 0$
$\mathcal{K}_\tau(A\ \mathtt{atprev}\ B) = $ t	iff	$\mathcal{K}_{\tau'}(A) = $ t for some $\tau' < \tau$
	and	$\mathcal{K}_{\tau'}(B) = $ t for the largest $\tau' < \tau$
	with	$\mathcal{K}_{\tau'}(A) = $ t

Statements like "Let $\tau' = \tau - 1$; we know $\tau' \rightarrow A$" can now by replaced by $\tau \rightarrow \triangle A$.

For proving facts like $\tau_s \rightarrow (B \texttt{ until } \tau_e)$ we will apply

Rule 5 (Bounded Induction) *In order to prove "if A holds at τ_s then B always holds from τ_s up to τ_e", it suffices to prove that B follows from A at τ_s and that B is at every τ true before τ_e under the assumption that it has been true from τ_s on up to τ:*

$$(\tau_s \rightarrow (A \rightarrow B)) \wedge \forall \tau_e > \tau > \tau_s((\tau_s \rightarrow (B \texttt{ until } \tau)) \rightarrow (\tau \rightarrow B)) \vdash$$
$$\tau_s \rightarrow (A \rightarrow (B \texttt{ until } \tau_e)).$$

This is a slight generalization of Rule 4 (Induction).

3.4 The Formal Model

3.4.1 The Overall Design

Our verification will be based on the following model program

$$\begin{aligned} \mathbf{\Pi} \ \equiv\ & \textbf{init } I \wedge J \\ & \textbf{cobegin } \Pi_1 \ \|\ \ldots\ \|\ \Pi_p \textbf{ coend} \end{aligned}$$

where each process Π_k has the structure

$$\begin{array}{ll} & \ldots \\ (\beta_1^k) & \mathbf{wait_k}(y_k, r_k, R_k) \\ (\beta_e^k) & \mathbf{nop} \\ & \ldots \\ (\alpha_1^k) & \mathbf{result_k}(x_k) \\ (\alpha_e^k) & \mathbf{stop} \end{array}$$

Each process may contain an arbitrary number of *wait* calls (with arbitrary local computations between these calls) and is terminated by a *result* call. The informal meaning of these functions is

1. $\mathbf{wait(y, r, R)}$ waits for the result of one of the Π_i where i is in the set R. After returning from the call, r holds one of these identifiers and y holds the result of Π_r.
2. $\mathbf{result(x)}$ makes the process result x public to all processes that currently wait for this result or will at a later time point try to retrieve the result.

The **nop** (no operation) statement after the *wait* call only serves as a reference to the last instruction of *wait*, the **stop** statements after *result* terminates the program.

Please note that our model only describes the synchronization of PACLIB programs by `pacWait` but not the creation of new processes by `pacStart`. However, we will never use the process number p in our verification; the proof also works for an infinite number of processes and therefore also for programs that create new processes at runtime.

Furthermore, we do not demand that all the Π_k execute the same program. In fact, we do not at all make any assumptions about the structure of the processes before or after a particular *wait* call. Our verification will therefore be valid for *any* set of processes that use the *wait* construct.

Before we describe the initialization conditions I and J and the internal structure of **wait** and **result**, we will formulate the correctness theorem that we are going to verify.

3.4.2 The Correctness Theorem

The correctness of the program is established by the following theorem.

Theorem 1 (Correctness) *For every process j that calls $wait_j(y_j, r_j, R_j)$ the following holds:*

1. **No Result:** *If no process i in R_j has ever executed or will ever execute $result_i$, then j will not return from the $wait_j$ call.*
2. **Some Result:** *If some process i in R_j has executed or will execute $result_i$ then the following holds:*
   (a) **Wait Continuation:** *j returns from the $result_j$ call.*
   (b) **Result Delivery:** *after returning from $result_j$, the variable r_j holds some identifier i of R_j and the variable y_j holds the corresponding result x_i.*

Formally,

$$
\begin{array}{l}
\forall j\ \forall \tau_w \geq \tau_\Pi\, \forall \overline{R}\ \forall \overline{x}\ (\\
\quad (\tau_\Pi \rightarrow \Box \forall i(\alpha_1^i \rightarrow x_i = \overline{x}_i)) \wedge (\tau_w \rightarrow (\beta_1^j \wedge R_j = \overline{R}_j)) \ \rightarrow \\
\qquad (\tau_\Pi \rightarrow \forall i \in \overline{R}_j(\Box \neg \alpha_1^i)) \ \rightarrow \\
\qquad\quad (\tau_w \rightarrow \Box \neg \beta_e^j) \ \wedge \\
\qquad (\tau_\Pi \rightarrow \exists i \in \overline{R}_j(\Diamond \alpha_1^i)) \ \rightarrow \\
\qquad\quad (\tau_w \rightarrow \Diamond \beta_e^j) \ \wedge \\
\qquad\quad (\tau_w \rightarrow ((r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j})\ \mathtt{atnext}\ \beta_e^j))).
\end{array}
$$

We will now explain this correctness theorem in more detail. The theorem contains the following mathematical constants and statement labels:

- j is the identifier of a process waiting for the result.
- τ_w is the time when j performs the $wait_j$ call.
- $\overline{R}_j$ is the "wait set" of j in this $wait_j$ call.
- $\overline{x}_i$ is the result of Π_i (for all i).
- β_1^j is the label of the $wait_j$ call.
- β_e^j is the label of the last statement of $wait_j$.
- α_1^i is the label of the $result_i$ call of Π_i (for all i).

The theorem formulates an assertion for each process j that performs a $wait_j$ call at some time point τ_w with some wait set $\overline{R}_j$.

The first assumption

$$\tau_\Pi \rightarrow \Box \forall i(\alpha_1^i \rightarrow x_i = \overline{x}_i)$$

declares the result of each process Π_i as the value of the program variable x_i at the time at which Π_i calls $result_i$. The result is associated to the mathematical constant $\overline{x}_i$, which is necessary because the value of x_i needs not be the same any more at the time when the $wait_j$ call returns the result of Π_i.

The second assumption

$$\tau_w \rightarrow (\beta_1^j \wedge R_j = \overline{R}_j)$$

defines τ_w to be the time at which Π_j calls $wait_j$ and associates the contents of the program variable R_j to the mathematical constant $\overline{R}_j$. This is required since the value of R_j may be changed during the program run.

The theorem then describes two cases:

1. $\boldsymbol{\tau_\Pi \rightarrow \forall i \in \overline{R}_j(\Box\neg\alpha_1^i)}$

   is the case when no process $i \in \overline{R}_j$ has ever called $result_i$ (i.e. *before* $wait_j$ is called) or will ever call $result_i$ (i.e. *after* $wait_j$ has been called). The reference time of the assumption is therefore the program start time τ_Π and the $\Box$ operator refers to the total runtime of the program. The conclusion

   $$\tau_w \rightarrow \Box\neg\beta_e^j$$

   says that Π_j will never return from this particular $wait_j$ call (but it does not say anything about former $wait_j$ calls of Π_j). The reference time of the conclusion is therefore the time τ_w when $wait_j$ is called and the $\Box$ operator only refers to the future of τ_w (but not to the time between τ_Π and τ_w).

2. $\boldsymbol{\tau_\Pi \rightarrow \exists i \in \overline{R}_j(\Diamond\alpha_1^i)}$

   is the case when some process $i \in \overline{R}_j$ has terminated before τ_w or will terminate after τ_w. The reference time of the assumption is therefore τ_Π and the $\Diamond$ operator refers to the total runtime of the program. The first conclusion

   $$\tau_w \rightarrow \Diamond\beta_e^j$$

   says that Π_j will return from this particular $wait_j$ call, i.e. it will execute β_e^j *after* τ_w. The second conclusion

   $$\tau_w \rightarrow ((r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j}) \ \texttt{atnext} \ \beta_e^j)$$

   says that after returning from this $wait_j$ call r_j holds one of the process identifiers of $\overline{R}_j$ and that y_j equals the result of this process. The reference time of this statement is therefore τ_w. It is not necessary to express by extra assertion that Π_{r_j} has terminated, since the only information about $\overline{x}_i$ can be taken from the assumption that associates $\overline{x}_i$ to the value of x_i at the termination time of Π_i.

Please note that the index i is always used if we refer to the role of a process Π_i as a deliverer of a result in $result_i$ while the index j is used if we refer to the role of some Π_j as a receiver of a result in $wait_j$. The details of $result_i$ and $wait_j$ are described in the next subsection.

3.4.3 The Detailed Structure

The following global variables are used in the model program for the communication and synchronization between processes:

- $value_i$ holds the result of Π_i after Π_i has terminated. This variable models the `value` field of a PACLIB result descriptor.
- $avail_i$ is a boolean flag signalling that $value_i$ holds the result of Π_i. This variable models the `avail` field of a PACLIB result descriptor.
- W_i is a set that holds the identifiers of all processes being blocked on an attempt to receive the result of Π_i. This variable models the queue defined by the `wait/value` fields of a PACLIB result descriptor.

- $wsem_i$ is a binary semaphore that provides mutual exclusion during the access to W_i. This variable models the `sem` field of a PACLIB result descriptor.
- R_j is the set that holds all the processes that some Π_j waits for. This variable one the one hand is the input argument of the $wait_j$ call, on the other hand it models the `wait` field of a PACLIB task descriptor.
- $rsem_j$ is a binary semaphore that provides synchronization between all Π_i that may deliver their result to Π_j. This variable models the `sem` field of a PACLIB task descriptor.
- run_j is a binary semaphore used to block a process Π_j when none of the Π_i with $i \in \overline{R_j}$ has yet terminated. This variable therefore models the dequeuing of a task descriptor from the scheduling queue.

This model is a simplification of the actual data representation in the PACLIB kernel, in particular concerning the following features:

1. **Data Structures:** The task queue in a PACLIB result descriptor and the task list in a PACLIB task descriptor are modelled by the mathematical sets W_i and R_j, respectively. Hence, modifications of queues and lists are modelled by set operations. This may essentially change the semantics of the program as discussed below.
2. **Result Descriptor:** The `value` field in a PACLIB result descriptor serves as the tail of the `wait` queue before it actually receives the result of the process. In the model program, this role is played by two distinct variables $value_i$ and W_i. This also has some crucial consequences on the program model as discussed below.
3. **Task Descriptor:** When a PACLIB task is blocked, the `wait` field of the task descriptor contains the set of tasks that the blocked task waits for. After it has been awaked, the `wait` field holds the identifier of the delivering task and the `value` field holds the result. Since in the model program all variables are globally known to all processes, Π_i directly reads R_j to determine the processes that Π_j is waiting for and writes itself i and x_i into r_j and y_j, respectively. We will ignore this issue in our verification.
4. **Life Time:** In the model program, all variables have an infinite life time and may be accessed even after some processes have terminated. In the PACLIB kernel, the local variables of a process and the contents of a task descriptor may be not accessed any more after the process has terminated since the process workspace has been freed and the contents of the corresponding memory cells are undefined. Since this problem is closely related to many low level details of process creation and termination, we will not deal with it in our verification.

Before we discuss the model in more detail, let us take a look at the internal structure of $result_i$ and $wait_j$ in Figure 3.5 and Figure 3.6, respectively.

- $result_i$ first writes the result x_i into the variable $value_i$ and sets the $avail_i$ flag. It then temporarily locks $wsem_i$ to receive one process $j \in W_i$ blocked on an attempt to receive the result of Π_i. Afterwards it locks $rsem_j$ and determines whether j is still in W_i or if another process $\Pi_{i'}$ has already delivered its result to Π_j and has removed j from W_i. If j is still in W_i, Π_i has "won the race" and iteratively removes j from the wait sets $W_{i'}$ of all $\Pi_{i'}$ that Π_j was waiting for. Afterwards, Π_i delivers its result to Π_j, awakes the task by calling $\mathbf{V}(run_j)$ and continues the delivery of results until all blocked task have been awaked.

$\mathbf{result_i}(x_i) \equiv$

(α^i_1) $value_i := x_i$
(α^i_2) $avail_i :=$ true

(α^i_3) $\mathbf{P}(wsem_i)$
(α^i_4) **while** $W_i \neq \emptyset$ **do**
(α^i_5) $w_i \in W_i$
(α^i_6) $\mathbf{V}(wsem_i)$

(α^i_7) $\mathbf{P}(rsem_{w_i})$

(α^i_8) $\mathbf{P}(wsem_i)$
(α^i_9) $deliver_i := w_i \in W_i$
(α^i_{10}) $\mathbf{V}(wsem_i)$

(α^i_{11}) **if** $deliver_i$ **then**

(α^i_{12}) $S_i := R_{w_i}$
(α^i_{13}) **while** $S_i \neq \emptyset$ **do**
(α^i_{14}) $s_i \in S_i$
(α^i_{15}) $\mathbf{P}(wsem_{s_i})$
(α^i_{16}) $\mathbf{rem}(W_{s_i}, w_i)$
(α^i_{17}) $\mathbf{V}(wsem_{s_i})$
(α^i_{18}) $S_i := S_i \backslash \{s_i\}$
endwhile

(α^i_{19}) $y_{w_i} := x_i$
(α^i_{20}) $r_{w_i} := i$
(α^i_{21}) $\mathbf{V}(run_{w_i})$

endif

(α^i_{22}) $\mathbf{V}(rsem_{w_i})$

(α^i_{23}) $\mathbf{P}(wsem_i)$
endwhile
(α^i_{24}) $\mathbf{V}(wsem_i)$

(α^i_e) **stop**

Figure 3.5: Result Process

$\mathbf{wait_j}(y_j, r_j, R_j) \equiv$

(β^j_1) $P_j := R_j$

(β^j_2) $\mathbf{P}(rsem_j)$
(β^j_3) **while** $P_j \neq \emptyset \wedge \neg found_j$ **do**

(β^j_4) $\quad p_j \in P_j$
(β^j_5) $\quad \mathbf{P}(wsem_{p_j})$

(β^j_6) $\quad$ **if** $\neg avail_{p_j}$ **then**

(β^j_7) $\qquad \mathbf{add}(W_{p_j}, j)$
(β^j_8) $\qquad \mathbf{V}(wsem_{p_j})$
(β^j_9) $\qquad P_j := P_j \backslash \{p_j\}$

$\quad$ **else**
(β^j_{10}) $\qquad \mathbf{V}(wsem_{p_j})$

(β^j_{11}) $\qquad y_j := value_{p_j}$
(β^j_{12}) $\qquad r_j := p_j$
(β^j_{13}) $\qquad found_j :=$ true

(β^j_{14}) $\qquad Q_j := R_j \backslash P_j$
(β^j_{15}) $\qquad$ **while** $Q_j \neq \emptyset$ **do**
(β^j_{16}) $\qquad\quad q_j \in Q_j$
(β^j_{17}) $\qquad\quad \mathbf{P}(wsem_{q_j})$
(β^j_{18}) $\qquad\quad \mathbf{rem}(W_{q_j}, j)$
(β^j_{19}) $\qquad\quad \mathbf{V}(wsem_{q_j})$
(β^j_{20}) $\qquad\quad Q_j := Q_j \backslash \{q_j\}$
$\qquad$ **endwhile**
$\quad$ **endif**

endwhile
(β^j_{21}) $\mathbf{V}(rsem_j)$

(β^j_{22}) **if** $found_j$ **then**
(β^j_{23}) $\quad found_j :=$ false
else
(β^j_{24}) $\quad \mathbf{P}(run_j)$
endif

(β^j_e) **nop**

Figure 3.6: Wait Process

- $wait_j$ first locks $rsem_j$ to prevent any process Π_i from delivering its result to Π_j while Pi_j is not yet blocked. Then Π_j iteratively examines the state of each $avail_i$ with $i \in R_j$. If the $avail_i$ flag is not yet set, Π_j adds j to the corresponding wait set W_i and continues the search. If none of the $avail_i$ flags is set, Π_j unlocks $rsem_j$ and blocks itself by calling $\mathbf{P}(run_j)$. After it has been awaked by some Π_i, the results are available by $r_j = i$ and $y_j = x_i$. However, if some of the Π_i has already delivered its result, Π_j itself sets r_j and x_j and removes j from all W_i to which it has added j before.

The algorithm itself reflects more or less the PACLIB program given in Section 3.2 with one major exception: while `pacWait` fills the `value` and the `avail` field of its result descriptor at the end of the program, the function $wait_j$ defines $value_j$ and $avail_j$ at the beginning. The reason for this is the following:

1. `pacWait` uses the `value` field to store the tail pointer of the `wait` queue before the result is available. Since other tasks may remove items from this queue as long as this queue is not empty, the result may be written into the `value` field only at the end of the function (when all blocked tasks have been processed).
2. It can *not* be proved that a $wait_j$ call always returns on the basis of a pure set interpretation of W_j if the $value_i$ variable is filled at the end of the program. We either have to add an axiom that asserts that W_j fulfills the "first in, first out" property of queues or restructure the program in the presented way.

We will discuss this issue in Section 3.5.3 in more detail. The last point that we have to mention is the definition of the operations **add** and **rem** that are used for operations on the sets W_j. In the PACLIB kernel, the `wait` queues in the PACLIB result descriptors are the only compound data structures that are modified by different tasks. The purpose of the semaphores in these descriptors is to guarantee the consistent modification of these queues.

However, in our program we have modelled the `wait` queues as sets W_j to which the set operations $\cup$ and $\setminus$ may be applied. However, these set operations are atomic and need no protection at all! Even, if we keep the protecting semaphores in the program and prove that they do not cause any harm (like deadlocks), we may and will apply the fact that sets can be atomically updated by one statement.

In order to make the model more realistic, we therefore replace the set operations $\cup$ and $\setminus$ on W_j by the functions **add** and **rem** defined by the following code sequences:

(λ_l^p) $\mathbf{add}(W, e) \equiv$

$(\lambda_{l_a}^p)$ $T_p := W$
$(\lambda_{l_b}^p)$ $W := \ldots$
$(\lambda_{l_c}^p)$ $W := T_p \cup \{e\}$

(λ_l^p) $\mathbf{del}(W, e) \equiv$

$(\lambda_{l_a}^p)$ $T_p := W$
$(\lambda_{l_b}^p)$ $W := \ldots$
$(\lambda_{l_c}^p)$ $W := T_p \setminus \{e\}$

where T_p is a new auxiliary variable. During the execution of these code sequences, there exists one *inconsistent* state during which we have no information on the value of W. If we are now able to verify the correctness of the model program, we may assume that the inconsistent state during the modifications of the `wait` queues does not cause any harm.

For all other sets, that are either only used by one process or not modified at all, we will use the mathematical set operations.

Finally, let us now define the initialization condition $I \wedge J$ of Π where I defines the initial states of all "*result*-related" program variables and J defines the initial states of all "*wait*-related" variables:

$$I \equiv \forall i(wsem_i \wedge \neg avail_i \wedge W_i = \emptyset)$$
$$J \equiv \forall j(rsem_j \wedge \neg found_j \wedge \neg run_j)$$

We assume that these variables will be only used within the functions $result_i$ and $wait_j$ and are in particular not modified elsewhere in Π. The main lines of the proof and the central theorems are presented in the next section.

3.5 The Verification

In this section, we will present the main lines of the verification of Theorem 1 (Correctness). The proof is based on four central theorems that can be informally described as follows:

1. **Theorem 2 (No Result)** says that the program flags $found_j$ and run_j will be never set to true, if no process Π with $i \in R_j$ terminates.
2. **Theorem 3 (Found)** says that if Π_j has set the found flag, it has itself found the result of some terminated process Π_i.
3. **Theorem 4 (Awake)** says that, if some Π_i has set run_j to true, Π_i has delivered its result to Π_j.
4. **Theorem 5 (Sometime)** says that some Π_i will eventually set run_j to true, if Π_j is blocked.

Please note that Theorem 2, Theorem 3 and Theorem 4 describe *invariance properties* while Theorem 5 describes an *eventuality property* of Π .

Theorem 2 is the basis of Proposition 1 (No Result) of the correctness theorem; Theorem 3 and Theorem 4 together establish the correctness of Proposition 2b (Result Delivery); last but not least Theorem 5 is the main constituent of Proposition 2a (Wait Continuation).

In the whole proof, in all theorems and lemmas, we assume that j, $\tau_w \geq \tau_\Pi$, $\overline{R}$ and $\overline{x}$ are arbitrary fixed such that

(3.1) $\tau_s \rightarrow \Box\forall i(\alpha_1^i \rightarrow x_i = \overline{x}_i)$,

(3.2) $\tau_w \rightarrow (\beta_1^j \wedge R_j = \overline{R}_j)$.

Furthermore, we will use the predicate

$$\mathtt{elem}_i(j, W) \equiv (wsem_i \rightarrow j \in W) \wedge (\neg wsem_i \rightarrow (j \in W \ \mathtt{atnext} \ wsem_i)).$$

which informally says that, if no process currently updates the set W, j is in W. However, if W is currently modified, then i will be in W when W is consistent again. In general, we use this predicate instead of $\in$ to describe properties of W_i because an assertion $j \in W_i$ that has held before W_i has been modified and will hold after the modification has been completed does not hold during the modification!

In the following three subsections, we will verify the three conclusions of the correctness theorem based on the theorems presented above. We will prove these theorems based on a set of lemmas described and proved in Appendix A. The reader should understand the contents of these lemmas (not their proofs) before reading the following subsections. The proofs of these lemmas actually represent the technically difficult parts of the verification.

3.5.1 No Result

We will prove Proposition 1 of Theorem 1 (Correctness). We assume that no process in $\overline{R}_j$ delivers a result

(3.3) $\tau_\Pi \rightarrow \forall i \in \overline{R}_j(\Box\neg\alpha_1^i)$

and want to prove that process j will not return from the $wait_j$ call

(3.4) $\tau_w \rightarrow \Box\neg\beta_e^j$.

By $\tau_\Pi \leq \tau_w$ and Axiom 3 (Execute), it suffices to prove

(3.5) $\tau_\Pi \rightarrow \Box\neg\,\mathtt{at}\{\beta_e^j\}$.

By Rule 4 (Induction), we first have to show

(3.6) $\tau_\Pi \rightarrow \neg\,\mathtt{at}\{\beta_e^j\}$

which is a direct consequence of $start_\Pi$ and Axiom 4 (Two Ready). Now let time $\tau > \tau_\Pi$ be arbitrary fixed. Assuming

(3.7) $\tau_\Pi \rightarrow (\neg\,\mathtt{at}\{\beta_e^j\}\ \mathtt{until}\ \tau)$

we have to prove

(3.8) $\tau \rightarrow \neg\,\mathtt{at}\{\beta_e^j\}$.

By the program structure and (3.7), we may assume

(3.9) $\tau \rightarrow \triangle(\beta_{23}^j \vee \beta_{24}^j)$.

since otherwise (3.8) is immediately fulfilled. We will now distinguish two cases:

Case $\boldsymbol{\tau \rightarrow \triangle\beta_{23}^j}$

By Theorem 2 (No Result), we have

(3.10) $\tau \rightarrow \triangle\neg found_j$.

This, by Lemma 32 (Reset), contradicts the case assumption.

Case $\boldsymbol{\tau \rightarrow \triangle\beta_{24}^j}$

By Theorem 2 (No Result), we have

(3.11) $\tau \rightarrow \triangle\neg run_j$.

This, by Axiom 11 (Wait Statement), contradicts the case assumption. $\Box$

We will now prove the central theorem of this subsection which says that, if no Π_i with $i \in R_j$ terminates, the flag $found_j$ and the binary semaphore run_j will be never set.

Theorem 2 (No Result) *If no process in $\overline{R}_j$ terminates, j will not find a result and be blocked forever i.e.*

$$(\tau_\Pi \rightarrow \Box\forall i \in \overline{R}_j(\neg\alpha_1^i)) \rightarrow (\tau_\Pi \rightarrow \Box(\neg found_j \wedge \neg run_j)).$$

Proof: Assume

(3.12) $\tau_\Pi \rightarrow \Box\forall i \in \overline{R}_j(\neg\alpha_1^i)$.

By Rule 4 (Induction), we first have to prove

(3.13) $\tau_\Pi \rightarrow (\neg found_j \wedge \neg run_j)$

which is a consequence of $start_\Pi$. Now let time $\tau > \tau_\Pi$ be arbitrary fixed. Assuming

(3.14) $\tau_\Pi \rightarrow ((\neg found_j \wedge \neg run_j)\ \mathtt{until}\ \tau)$

we have to prove

(3.15) $\tau \rightarrow (\neg found_j \wedge \neg run_j)$.

By the structure of the program, we may assume

(3.16) $\tau \rightarrow \triangle(\beta^j_{13} \vee \exists i(\alpha^i_{21} \wedge w_i = j))$.

since these are the only statements modifying $found_j$ and run_j. However, $\tau \rightarrow \triangle\beta^j_{13}$ contradicts Lemma 26 (If Available) and Lemma 27 (Not Available). Furthermore, $\tau \rightarrow \triangle\exists i(\alpha^i_{21} \wedge w_i = j)$ contradicts by the program structure (3.12). □

3.5.2 Result Delivery

We will prove Proposition 2b of Theorem 1 (Correctness). We assume

(3.17) $\tau_\Pi \rightarrow \exists i \in \overline{R}_j(\Diamond\alpha^i_1)$.

and want to prove

(3.18) $\tau_w \rightarrow ((r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j})\ \mathtt{atnext}\ \beta^j_e)$.

In case of

(3.19) $\tau_w \rightarrow \Box\neg\beta^j_e$

we are done. Otherwise, by the program structure there exist times τ_d, τ_e, τ_f, τ_r such that

(3.20) $\tau_w \rightarrow (\tau_d\ \mathtt{atnext}\ \beta^j_2)$,

(3.21) $\tau_w \rightarrow (\tau_e\ \mathtt{atnext}\ \beta^j_{21})$,

(3.22) $\tau_w \rightarrow (\tau_f\ \mathtt{atnext}\ \mathtt{at}\{\beta^j_e\})$,

(3.23) $\tau_w \rightarrow (\tau_r\ \mathtt{atnext}\ \beta^j_e)$.

These definitions will be used in this section and all associated lemmas. We now can reformulate (3.18) as

(3.24) $\tau_r \rightarrow (r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j})$.

By Axiom 3 (Execute), we know

(3.25) $\tau_r \rightarrow \mathtt{at}\{\beta^j_e\}$.

We will prove (3.24) by induction on the number t of times that β^r_e was not scheduled for execution i.e. on $t = \tau_r - \tau_f$. The base case of the induction $\tau_r = \tau_f$ characterized by

(3.26) $\tau_r \rightarrow \triangle(\beta^j_{23} \vee \beta^j_{24})$

will be shown in the following two subsections. For the induction step, let us assume

(3.27) $\tau_r \rightarrow \triangle(\mathtt{at}\{\beta^j_e\} \wedge r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j})$.

By Lemma 46 (Exit), we also know

(3.28) $\tau \rightarrow \triangle\forall i(\mathtt{at}\{\alpha^i_6 \ldots \alpha^i_{21}\} \rightarrow w_i \neq j)$

i.e. r_j and y_j cannot be modified by any Π_i. Therefore we have (3.24).

We will further on deal with the two cases of the induction base.

Case $\tau_r \rightarrow \triangle\beta^j_{23}$

By Lemma 32 (Reset), we know

(3.29) $\tau_r \rightarrow \triangle found_j$.

By Theorem 3 (Found), we then know

(3.30) $\tau_r \rightarrow \triangle(r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j})$.

By the program structure, we then have (3.24).

We will now prove the first central theorem of this subsection which says that if Π_j has set the $found_j$ flag it has found the result of some Π with $i \in R_j$.

Theorem 3 (Found) *If j has set the found flag, it has indeed found a result:*

$$\tau_w \rightarrow ((found_j \rightarrow (r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j})) \texttt{ until } \tau_r).$$

Proof: Define

(3.31) $E \equiv found_j \rightarrow (r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j})$.

By Rule 5 (Bounded Induction), we first have to show

(3.32) $\tau_w \rightarrow E$

which is a consequence of Lemma 35 (Entry). Now let τ be arbitrary fixed with $\tau_w < \tau < \tau_r$ and assume

(3.33) $\tau_w \rightarrow (E \texttt{ until } \tau)$.

We have to prove

(3.34) $\tau \rightarrow E$.

We may assume

(3.35) $\tau \rightarrow \triangle(\beta^j_{11} \vee \beta^j_{12} \vee \beta^j_{13} \vee \exists i(w_i = j \wedge (\alpha^j_{19} \vee \alpha^j_{20})))$

since these are the only statements modifying y_j, r_j or $found_j$.

Case $\tau \rightarrow \triangle(\beta^j_{11} \vee \beta^j_{12})$

By Lemma 31 (Not Found), we have

(3.36) $\tau \rightarrow \triangle\neg found_j$

and hence, by the program structure,

(3.37) $\tau \rightarrow \neg found_j$.

Case $\tau \rightarrow \triangle\beta^j_{13}$

By Lemma 26 (If Available), we know

(3.38) $\tau \rightarrow \triangle avail_{p_j}$

and by Lemma 21 (Result Exclusion)

(3.39) $\tau \rightarrow \triangle(r_j = p_j)$.

This implies by Lemma 28 (Process Set) and Lemma 29 (Result)

(3.40) $\tau \rightarrow \triangle(r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j})$.

Then we are by the program structure done.

Case $\tau \rightarrow \triangle(\mathbf{w_i = j} \wedge (\alpha^{\mathbf{i}}_{\mathbf{19}} \vee \alpha^{\mathbf{i}}_{\mathbf{20}}))$

By Lemma 38 (Lock), we know

(3.41) $\tau \rightarrow (\mathtt{elem}_i(j, W_i)\ \mathtt{atprev}\ \alpha^i_7)$.

Furthermore, by Axiom 11 (Wait Statement) we know

(3.42) $\tau \rightarrow (rsem_j\ \mathtt{atprev}\ \alpha^i_7)$.

This implies by Lemma 44 (Result Loop)

(3.43) $\tau \rightarrow ((\neg found_j \wedge \neg run_j \wedge\ \mathtt{at}\{\beta^j_{22}, \beta^j_{24}\})\ \mathtt{atprev}\ \alpha^i_7))$.

By Lemma 21 (Result Exclusion) and the program structure we then know

(3.44) $\tau \rightarrow \neg found_j$. □

Case $\tau \rightarrow \triangle\beta^{\mathbf{j}}_{\mathbf{24}}$

By Axiom 11 (Wait Statement), we know

(3.45) $\tau \rightarrow \triangle run_j$.

Thus we know by Theorem 4 (Awake)

(3.46) $\tau \rightarrow \triangle(r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j})$

By the program structure, we then have (3.24). □

We will now prove the second central theorem of this subsection which says that if some Π_i sets run_j it has delivered its result to Π_j.

Theorem 4 (Awake) *If process j is waked up by some process i, then j has received the result x_i:*

$$\tau_w \rightarrow ((run_j \rightarrow (r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j}))\ \mathtt{until}\ \tau_r).$$

Proof: Define

(3.47) $F \equiv run_j \rightarrow (r_j \in \overline{R}_j \wedge y_j = \overline{x}_{r_j})$.

By Rule 5 (Bounded Induction), we first have to prove

(3.48) $\tau_w \rightarrow F$

which is true since Lemma 35 (Entry) implies $\tau_w \rightarrow \neg run_j$. Let τ be arbitrary fixed with $\tau_w < \tau < \tau_r$ and assume

(3.49) $\tau_w \rightarrow (F\ \mathtt{until}\ \tau)$.

We have to prove

(3.50) $\tau \rightarrow F$.

We may assume

(3.51) $\tau \rightarrow \triangle(\beta^j_{11} \vee \beta^j_{12} \vee \exists i(w_i = j \wedge (\alpha^i_{19} \vee \alpha^i_{20} \vee \alpha^i_{21})))$

since these are the only statements that may modify y_j, r_j or run_j.

Case $\tau \rightarrow \triangle(\beta^{\mathbf{j}}_{\mathbf{11}} \vee \beta^{\mathbf{j}}_{\mathbf{12}})$

By $\tau < \tau_e$, Lemma 39 (Not Run) implies

(3.52) $\tau \to \triangle \neg run_j$.

By the program structure, we thus have

(3.53) $\tau \to \neg run_j$.

Case $\boldsymbol{\tau \to \triangle(\mathbf{w_i = j} \wedge (\alpha^{\mathbf{i}}_{\mathbf{19}} \vee \alpha^{\mathbf{i}}_{\mathbf{20}}))}$

Lemma 38 (Lock) implies

(3.54) $\tau \to (\texttt{elem}_i(j, W_i)\ \texttt{atprev}\ \alpha^i_7)$.

By Axiom 11 (Wait Statement), we know

(3.55) $\tau \to (rsem_j\ \texttt{atprev}\ \alpha^i_7)$.

By Lemma 44 (Result Loop), we thus know

(3.56) $\tau \to (\neg run_j\ \texttt{atprev}\ \alpha^i_7)$.

This implies by Lemma 21 (Result Exclusion)

(3.57) $\tau \to \neg run_j$.

Case $\boldsymbol{\tau \to \triangle(\mathbf{w_i = j} \wedge \alpha^{\mathbf{i}}_{\mathbf{21}})}$

By the structure of the program and Lemma 21 (Result Exclusion), we know

(3.58) $\tau \to \triangle(r_j = i \wedge y_j = \overline{x}_i)$.

By Lemma 38 (Lock), we then know

(3.59) $\tau \to (\texttt{elem}_i(j, W_i)\ \texttt{atprev}\ \alpha^i_7)$.

which implies by Lemma 36 (Membership)

(3.60) $\tau \to (i \in \overline{R}_j\ \texttt{atprev}\ \alpha^i_7)$

and thus by Lemma 24 (Constancy)

(3.61) $\tau \to \triangle(i \in \overline{R}_j)$.

This implies by the program structure

(3.62) $\tau \to (r_j \in \overline{R}_j \wedge y_j = \overline{x}_j)$. $\square$

3.5.3 Wait Continuation

We are going to prove Proposition 2a of Theorem 1 (Correctness). We assume

(3.63) $\tau_\Pi \to \exists i \in \overline{R}_j(\Diamond\, \texttt{at}\{\alpha^i_1\})$

and have to prove

(3.64) $\tau_w \to \Diamond\, \texttt{at}\{\beta^j_e\}$.

By Lemma 47 (Wait Continuation) and Lemma 51 (Result Continuation), we know

(3.65) $\tau_w \to \Diamond \beta^j_{21}$.

We thus may define some time $\tau_e > \tau_w$ such that

(3.66) $\tau_w \to (\tau_e\ \texttt{atnext}\ \beta^j_{21})$.

We may assume

(3.67) $\tau_e \rightarrow \neg found_j$

since otherwise, by Lemma 17 (Locality) and the program structure, we are done. Thus we know

(3.68) $\tau_e \rightarrow \Diamond\,\mathtt{at}\{\beta^j_{24}\}$

and have to show

(3.69) $\tau_e \rightarrow (\Diamond run_j \ \mathtt{atnext}\ \beta^j_{24})$.

From (3.67) we know by Lemma 42 (Wait End)

(3.70) $\tau_e \rightarrow \forall i \in \overline{R}_j(\mathtt{elem}_i(j, W_i))$

By the program structure, we also know

(3.71) $\tau_e \rightarrow \bigcirc rsem_j$

which implies by Theorem 5 (Sometime)

(3.72) $\tau_e \rightarrow \bigcirc\Diamond run_j$

and, since run_j is nowhere else set to false, (3.69). □

The following theorem says that, if Π_j has put j into some W_i, some Π_i will eventually set run_j.

Theorem 5 (Sometime) *If j is in some W_i, run_j will be eventually set.*

$$\tau_w \rightarrow (((rsem_j \wedge \exists i(\mathtt{elem}_i(j, W_i))) \rightarrow \Diamond run_j)\ \mathtt{until}\ \tau_r).$$

Proof: Let $\tau \geq \tau_w$ be arbitrary fixed and assume for some i

(3.73) $\tau \rightarrow (rsem_j \wedge \mathtt{elem}_i(j, W_i))$.

By Lemma 37 (Waiting), we know

(3.74) $\tau \rightarrow \mathtt{at}\{\beta^j_{22} \ldots \beta^j_e\}$.

By Lemma 54 (Result Induction), we know

(3.75) $\tau_e \rightarrow \Diamond(\alpha^i_4 \wedge W_i = \emptyset)$.

By (3.74) and the program structure, we know for some $i' \in \overline{R}_j$

(3.76) $\tau_e \rightarrow \Diamond(\alpha^{i'}_{16} \wedge s_{i'} = i \wedge w_{i'} = j)$.

This implies

(3.77) $\tau_e \rightarrow \Diamond(\alpha^{i'}_{21} \wedge w_{i'} = j)$

and we are by the program structure done. □

The proof of the previous theorem is essentially based on Lemma 54 (Result Induction) saying that $result_i$ always terminates. This lemma holds since $result_i$ defines $value_i$ and sets the flag $avail_i$ i.e. *before* it starts to deliver its result to other tasks. Please note that this is not true in the actual implementation of `pacWait` where $avail_i$ is set *after* the result has been delivered. However, if we add the following axiom that demands a *queue* interpretation of W_i, Theorem 5 (Sometime) is true again.

Axiom 18 (Queue) *W_i is accessed in a first in/first out manner:*

$$\tau_\Pi \rightarrow \Box\forall i \forall j \forall j'((\mathtt{elem}_i(j, W_i) \wedge \neg\mathtt{elem}_i(j', W_i)) \rightarrow (\neg\mathtt{elem}_i(j, W_i) \ \mathtt{before} \ \neg\mathtt{elem}_i(j', W_i))).$$

By this axiom, we may use Lemma 55 (First Out) to verify Theorem 5 (Sometime):

Proof: Let $\tau \geq \tau_\Pi$ be arbitrary fixed and assume for some i

(3.78) $\tau \rightarrow (rsem_j \wedge \mathtt{elem}_i(j, W_i))$.

By Lemma 37 (Waiting), we know

(3.79) $\tau \rightarrow \mathtt{at}\{\beta^j_{22} \ldots \beta^j_e\}$.

By Lemma 55 (First Out), we know

(3.80) $\tau \rightarrow \Diamond\neg\mathtt{elem}_i(j, W_i)$.

By (3.79) and the program structure, we know for some $i' \in \overline{R}_j$

(3.81) $\tau_e \rightarrow \Diamond(\alpha^{i'}_{16} \wedge s_{i'} = i \wedge w_{i'} = j)$.

This implies

(3.82) $\tau_e \rightarrow \Diamond(\alpha^{i'}_{21} \wedge w_{i'} = j)$

and we are by the program structure done. $\Box$

3.5.4 Proof Techniques

We conclude the verification by summarizing the basic proof techniques that we have used. For the semi-formal reasoning process, we usually applied the generalized induction rule introduced in Subsection 3.3.6.

1. **Locality:** If parts of a program only operate on local variables (i.e. variables that are only modified by the current process), these program parts can be verified without considering the existence of other processes. The complexity of the verification is thus essentially decreased.
2. **Program Variables:** Whenever possible we have expressed program properties in terms of program variables instead of program locations. For instance, Theorem 3 (Found) and Theorem 4 (Awake) are formulated using the variables $found_j$ and run_j and not the program locations β^j_{23} and β^j_{24}, respectively. Thus the propositions become more easy to understand and express the intuitions of the programmer better than by dealing with program locations. This is especially of advantage when code blocks are rearranged such that the locations of statements change but the proposition in terms of program variables still holds.
3. **Critical Regions:** The purpose of a critical region C

   (λ_a) $\mathbf{P}(sem)$
   C
   (λ_b) $\mathbf{V}(sem)$

   is described by a proposition of the form

   $$\Box(sem \rightarrow A)$$

   saying that A holds when no process is within C and therefore also when a process enters respectively exits C. For proving above formula, we have to prove $\tau_\Pi \rightarrow A$ and, for some arbitrary fixed time $\tau > \tau_\Pi$,

   $$\tau \rightarrow (\lambda_b \rightarrow \bigcirc A)$$

i.e. that the A holds when C is left. For the proof of this proposition, one may assume the induction hypothesis

$$\tau \rightarrow (A \texttt{ atprev } \tau_a)$$

i.e. that A was true when C was entered. The reasoning from τ_a to τ_e may be based on the fact that no other process is in C. Furthermore, we have to prove for each statement λ in every other process

$$(\lambda \wedge sem \wedge A) \rightarrow \bigcirc A$$

where we may restrict our consideration to those λ that might change the truth value of A (e.g. by modifying some program variable occurring in A).

4. **While Loops:** The invariant of a loop

   ```
   (λ')   ...
   (λ)    while B do
             ...
   (λ'')     ...
          endwhile
   ```

   is expressed by a proposition of the form

   $$\Box(\texttt{ at}\{\lambda\} \rightarrow A)$$

   which says that A always holds when execution is at the location λ. For proving this formula, we have to prove

   $$\Box(\lambda' \rightarrow \bigcirc A)$$

   i.e. that A holds when the loop is entered for the first time and

   $$(\texttt{ at}\{\lambda\} \wedge \neg\lambda \wedge A) \rightarrow \bigcirc A$$

   i.e. A does not become false if some other process is executed (where we may restrict considerations to those statements in other processes that may affect A) and

   $$\lambda'' \rightarrow \bigcirc A$$

   i.e. A holds after each iteration. For proving this fact, we may apply the induction hypothesis

   $$\lambda'' \rightarrow (A \wedge B \texttt{ atprev } \lambda)$$

   and then reason from one execution of λ to the next.

5. **Nested Loops:** The invariants A_1 and A_2 of two nested loops

   ```
   (λ1)   while B1 do
             ...
   (λ2)      while B2 do
                ...
             endwhile
             ...
          endwhile
   ```

   can be only verified by forming a joint invariant

   $$A \equiv (\texttt{ at}\{\lambda_1\} \rightarrow A_1) \wedge (\texttt{ at}\{\lambda_2\} \rightarrow A_2)$$

   and proving this joint proposition at all interesting transitions. A may be assumed for all former time moments (similar to the case of a single while loop).

6. **Critical Loops:** If the entry of a loop is part of a critical region

$$
\begin{array}{ll}
 & \mathbf{P}(sem) \\
(\lambda) & \textbf{while } B \textbf{ do} \\
 & \quad \ldots \\
 & \quad \mathbf{V}(sem) \\
 & \quad \ldots \\
 & \quad \mathbf{P}(sem) \\
 & \textbf{endwhile} \\
 & \mathbf{V}(sem)
\end{array}
$$

   the loop invariant actually reduces to the invariant of the critical region protected by *sem* extended by the proposition B i.e.

   $$\Box(sem \wedge B \rightarrow A)$$

   and then proceeds in a way similar to simple while loops.

7. **Continuation:** A good strategy to verify the continuation of a program is to verify the freedom of deadlocks of critical regions and the termination of loops from inside to the outside starting with the innermost region/loop. Parallel programs are usually constructed in such a way that there exists a partial ordering $\prec$ on semaphores such that $s_1 \prec s_2$ implies that a critical region protected by s_1 is only entered if the process is not in a critical region protected by s_2. This ordering also guides the structure of the proof.

There can be no general recipe how the verification of a parallel program with shared variables has to be organized to be manageable but also convincing. In the worst case, all parallel processes may be coupled so tightly that the full calculus of temporal logic has to be applied by constructing for every statement in every process an assertion that describes the full state of the program. Each of these assertions has then to be verified against every statement.

However, such situations probably occur very rarely. In many cases, the verification style presented in this chapter may save a considerable amount of work and make the verification a valuable documentation of the program. A high-level parallel programming model should be constructed in such a way that the model is powerful enough to express the parallel algorithms one is interested in but also restricted enough to cut down the complexity of parallel program verification.

3.6 The Role of Parallel Program Verification

In this section, we will present the conclusions with have drawn from this case study about the formal verification of parallel programs, its usefulness and its limitations. They can be summarized as

1. The goal of parallel program verification is to increase the confidence in some properties of an executable parallel program.
2. However, the formal verification process is only able to deal with a reduced model of this program.
3. The adequacy of the model with respect to the interesting properties of the real program cannot be checked by formal methods.

4. Even a formal verification of the model in general extremely complex and only feasible by radical simplifications.

5. Parallel program verification is therefore as error-prone as parallel programming itself and its adequacy cannot be guaranteed.

6. The importance of parallel program verification is that of documenting a parallel program more than of proving it.

3.6.1 Organization of the Verification

The process of program verification consists in our opinion of the following components:

1. **A Running Program:** There is given a program p that has in all past test runs shown the desired behavior. However, the programmer would like to increase his confidence that the program will show this behavior for all future runs (and all given inputs) and/or convince other people of this fact. For this purpose, informal reasoning in a natural language is by its ambiguity not considered sufficient.

2. **A Formal Framework:** The programmer chases a framework that allows more precisely than natural language to express properties of the program and to reason about these properties. To simplify the reasoning, this framework is a reduced image of the "real world" (i.e. computer hardware, operating system, programming language) neglecting all effects that the programmer considers of minor importance for the aspects of the program he is interested in. By the formal structure of the model, the correctness of the reasoning process can be more easily controlled.

3. **Program Model:** In the chosen framework, the programmer constructs a model $\bar{p}$ of the program and a proposition $c_{\bar{p}}$ that describes the notion of "correctness" in this model. If for each possible "model run" r of $\bar{p}$ the proposition $c_{\bar{p}}(r)$ holds, the program is considered correct. If a single counterexample r' with $\neg c_{\bar{p}}(r')$ can be constructed, the program is considered incorrect. However, the question whether $\bar{p}$ and $c_{\bar{p}}$ adequately describe p and its effects in the "real world" is not a question that can be answered by formal methods.

4. **Correctness Proof:** In the chosen framework, the programmer argues why the property $\forall r(c_{\bar{p}}(r))$ is true. Guided by some proof strategy, he reduces this proposition to other propositions whose correctness can be verified more easily. Ideally, the reasoning process is carried to an end where the propositions and the reasoning becomes so simple that it can be automatically checked by a computer. Practically, the reasoning process is stopped at a level where the correctness of the propositions is considered "obvious" for a human being.

3.6.2 Program Model and Implementation

The organization of program verification presented above contradicts the common view of the matter concerning the role of p and $\bar{p}$, respectively:

- In the common view, $\bar{p}$ is considered as the *algorithm* and the central point of discussion and p is considered as the *implementation* of $\bar{p}$.

- In our view, $\bar{p}$ is a *model* of p. Actually, we are interested in the properties of the program p and have introduced $\bar{p}'$ to make the domain of discourse tractable.

The main difference between both points of view is the importance that is given to the "mapping" of p to $\bar{p}$ or vice versa, respectively. Many promoters of the common view consider the design of p (if they are at all aware of the difference between p and $\bar{p}$) as a tedious but actually not very interesting task (since p is just the implementation of the $\bar{p}$ whose correctness has already been verified).

This view has its origin in the fact, that formal frameworks for sequential algorithms very well capture the main properties of real programs one is usually interested in. These properties are whether the program terminates and delivers the correct result for any valid input (and how long the computation takes). For an adequate description of these properties, many details of the program may be neglected. Most important, the model needs not contain any notion of *time relations*, since a program run can be easily modelled as a static sequence of variable assignments.

However, in parallel programming the gap between an abstract model and a concrete implementation is often very wide; a lot of textbooks on parallel algorithms use a notation that is far away from the constructs in which parallel programs have to be actually implemented. Many properties of the concrete program p can then not at all be derived from properties of the the abstract model $\bar{p}$, since these properties essentially depend on aspects that are usually decided at the time of the implementation only. If we only construct an abstract model $\bar{p}$ and verify its correctness, it is easily possible that the model will be to simple to capture the essentials of the concrete p.

Of course, before a programmer starts implementing a parallel program, he has designed some abstract algorithm $\bar{p}$ and convinced himself about the correctness of the ideas expressed in $\bar{p}$. However, as explained above, the actual implementation p will usually have a significant feedback also on $\bar{p}$ making the model more realistic and useful. Therefore we think that a prototype implementation p should precede a formal verification of $\bar{p}$. This has also the advantage that by testing p, many "trivial" errors in $\bar{p}$ (like wrong boundary conditions) can be found in an early stage far more easily than by a formal verification.

3.6.3 The Complexity of Verification

But if $\bar{p}$ is only a simplified model of p, are we then at least able to guarantee the correctness of $\bar{p}$ within the formal framework? Before we discuss the question, we would like to draw attention to the issue of the complexity of program verification.

Verifications of sequential programs are *compositional* in the sense that properties of independent program parts can be independently verified. For instance, in order to prove

$$\{a\}P;Q\{b\}$$

in Hoare's calculus, it suffices to prove for some proposition c

$$\{a\}P\{c\} \;\wedge\; \{c\}Q\{b\}$$

where both proofs can be carried through independently of each other. The verification of a large program can therefore be neatly decomposed into verifications of independent modules; the complexity of the verification reflects the complexity of the program structure.

In the case of parallel programs, the situation is fundamentally different. The proof of

$$\{a\}P||Q\{b\}$$

can not be decomposed into independent proofs on P and Q, since each operation in P may in general have some effect on the execution of Q. Instead, we have to show that if a

initially holds, b holds after the execution of the program *for any possible interleaving order* of the program operations.

Of course, if the processes contain sequential modules that do not interact with other processes, these modules may be considered as "atomic" and their correctness can be independently checked by sequential verification methods. However, the complexity of the proof grows in a combinatorial way with the number of interactions and the number of processes involved in these interactions. Therefore, for tightly coupled parallel processes with complex interaction patterns a fully formal verification is in general not feasible. We consider this as a principal limit to formal program verification that can not be overcome no matter which formal framework is applied.

There are two possibilities to cut down the complexity of the verification to a tractable order. The first one is to use parallel programming languages that restrict the interaction between parallel processes to particular patterns for which simple formal frameworks can be constructed (see Subsection 3.6.5). Even if the parallel program is developed in a language that allows very general forms of interactions, many simplifications can be performed, if these general forms are not really applied. However, then the verification also has to include the proof that the simplifications are really valid, which may be a very complex task in its own.

For parallel programs where the general forms of interaction are actually applied, the only chance to make a verification manageable is to let the human brain serve as a "preprocessor" performing radical simplifications by investigating only the "interesting" cases and by combining several inference steps into one using "higher-order" verification rules. However, the justification of these simplifications itself is then necessarily an informal one and prone to human errors. The persuasion of the proof decreases up to some point where the verification itself becomes less convincing than the fact that the program p has shown the correct behavior for a finite amount of times.

The saying "testing is only able to detect errors, the absence of errors can be only guaranteed by a formal verification" seems therefore questionable. Verifications are not just "right" or "wrong" (this could be only checked by a computer if the verification was fully formal), they are more or less "convincing" for human beings! The whole process of parallel program verification is a steady tradeoff between the two extremes of simplicity and significance, between readable proofs that are not very persuasive and convincing proofs whose details nobody is able to follow.

These arguments are of course also true for the verification of sequential programs or (to some extent) for the proof of any mathematical properties. However, in the case of parallel program verification they become especially striking, since the principal complexity of the verification process itself is much larger.

3.6.4 The Purpose of Verification

What is then the use of parallel program verification? A fully formal verification is in general not tractable, it is uncertain whether the program model captures all important effects and the proof itself may contain large informal inference steps. A formal program verifications is therefore at least as prone to errors as the process of programming itself.

Actually we consider as the main value of the verification of a parallel program not the aspect of *proving* (in the sense of establishing without doubt) the correctness of a program but the aspect of *documenting* the programmer's knowledge by

1. formal propositions that he believes to capture the essential properties of the program and by
2. (semi-)formal arguments that support the believe in the correctness of these propositions.

By a program verification, the programmer is (by some proof strategy) forced to methodologically think about the program and to formulate his thoughts, his knowledge and his opinions as formal propositions and (semi-)formal proofs. By this process itself, many errors can be captured and the confidence in the correct behavior of a program can be considerably increased. The result of this process, the propositions and the proof, may then serve as a documentation of the program and as a trace of the programmer's thoughts. Moreover, by the formal notation, the propositions can be irrefutably falsified by anybody that is able to construct counterexamples.

3.6.5 The Role of the Parallel Programming Model

The parallel programming model (respectively language) for implementing p may have some important impact on the verification process. The arguments concerning the complexity of parallel program verification presented in the previous subsections mainly applied to the classical parallel programming models with sequential processes coupled by a global store (shared memory programming) or special communication channels (message passing programming). The more tightly coupled the parallel processes are, the more complex their interaction and therefore their verification becomes.

On the other hand, parallel models based on concepts such as functional programming are much more easy to understand. The only interaction between processes occurs at the time of process creation (when arguments are passed) and process termination (when the result is delivered). For such models, no special parallel verification techniques are required, since here the correctness of the parallel program is a direct consequence of the correctness of the sequential program.

The parallel programming model of PACLIB is a compromise between both extremes. If only the deterministic construct `pacWaitOne` is applied for communication between processes, the correctness of a PACLIB program directly follows from the correctness of the corresponding SACLIB program where all `pacStart` and `pacWaitOne` constructs are removed. Even if the general non-deterministic `pacWait` construct is used, program verification is fairly easy.

Only, if processes are coupled by shared memory and semaphores (or by messages in the distributed memory case), the verification of programs becomes extremely complex. The purely functional model of parallelism should therefore be preferred, since the resulting programs are by orders of magnitude far more easy to understand and to verify.

Chapter 4

The Functional Language

In this chapter, we describe the design and implementation of D, a small functional language with strict execution semantics. Our goal is not to develop yet another functional language with many new concepts but to have a sound and efficient basis for the work described in Chapter 6.

The emphasis of the presentation is therefore less on the definition of the language but on its *implementation*:

1. **Target Code.** The compiler generates optimized C target code whose quality and structure essentially matches the manually written C code from which the SACLIB library [Buchberger *et al.*, 1992] is constructed. Special emphasis is given to the efficient processing of list structures.
2. **Runtime System.** The target code does not depend on a particular runtime system but can be linked with any kernel that supports the allocation of lists/tuples and automatic memory management (garbage collection).
3. **External Interface.** The target code generated from a D program can be called from external C programs. Vice versa, the functional program can call external C functions and procedures and use externally defined objects. Modules implemented in C and modules implemented in D can be arbitrarily combined in application programs.

The rest of the chapter is organized as follows:

- Section 4.1 gives an overview on D and formally defines a subset of it (the *core language*).
- Section 4.2 describes the concept of *iterated compositions*, the main original contribution of the language itself. This construct forms the basis of the parallel compositions described in Chapter 7.
- Section 4.3 describes how a D program is translated into a program in the core language by a set of transformations.
- Section 4.4 describes how optimized C target code is generated from a program in the core language.
- Section 4.5 defines the external interface between C modules and D modules.
- Section 4.6 gives some examples and benchmark results.

This chapter is mainly taken from [Schreiner, 1993a].

4.1 The Core Language

The D language provides the following features:

1. *Referential Transparency:* A D program is an expression whose evaluation does not cause any side effects. Consequently, there exist simple rules for reasoning and manipulation which are exploited by our compiler.
2. *Strict Semantics:* D has a strict evaluation semantics i.e. the arguments of a function can be evaluated before the function is called. This allows an efficient implementation of the language and a simple reasoning about its resource requirements.
3. *Polymorphic Type System:* The language allows to define functions that operate on objects of various types of a given pattern. The compiler translates a polymorphic function into a set of monomorphic ones.
4. *Function Templates:* D allows a limited class of higher-order functions, namely functions that are parameterized by other functions. Such functions are instantiated at compile-time yielding various first-order functions.
5. *Iterated Compositions:* D provides a flexible iteration construct that allows to express in a concise way (without applying recursive functions) frequently occurring computational patterns.

In this section we formally define the semantics of a core of this language. Sections 4.2, 4.3, and 4.5 describe how the remaining features of D are translated into this core language.

For the formal language definition, we essentially apply the denotational framework described in [Schmidt, 1986].

4.1.1 Abstract Syntax

Figure 4.1 gives the abstract syntax of the core language. The abstract syntax does *not* necessarily correspond to the concrete textual syntax of the full language (which is patterned after the syntax of HASKELL [Hudak *et al.*, 1992] and is given in Appendix B) but rather reflects the structure of the parse tree in the compiler. Keywords are only used as syntactic sugar to improve legibility of the definition.

This abstract syntax reflects the structure of the program after several phases of program transformations:

1. *Uncurrying:* Each partial application of a function (with less actual arguments than formal parameters) has been replaced by the identifier of a new function whose arity is the number of missing arguments.
2. *Specialization:* Each application of a higher-order function has been transformed into the application of an instance of this function by updating the formal parameter function with the actual argument function.
3. *Monomorphization:* Each application of a polymorphic function has been transformed into the application of a monomorphic instance of this function.
4. *Lambda Lifting:* All function definitions have been transformed into combinators (i.e. functions that do not reference the global environment) and lifted to the top-level.
5. *Type Annotations:* Type definitions and signatures have been removed and each function definition has been annotated by its type. We also assume all constant definitions to be annotated (just for uniformity, see Subsection 4.1.7).

$$
\begin{array}{llll}
\mathit{program} & \rightarrow & \texttt{let}\ \mathit{fdefs}\ \texttt{in}\ \mathit{exp} & \text{Function Definitions} \\
 & & & \\
\mathit{exp} & \rightarrow & \texttt{let}\ \mathit{cdefs}\ \texttt{in}\ \mathit{exp} & \text{Constant Definitions} \\
 & \mid & \texttt{if}\ \mathit{exp}_c\ \texttt{then}\ \mathit{exp}_t\ \texttt{else}\ \mathit{exp}_f & \text{Conditional} \\
 & \mid & \texttt{case}\ \mathit{exp}_s\ \texttt{of}\ \mathit{patterns} & \text{List Selection} \\
 & \mid & \mathit{exp}_h\ \texttt{:}\ \mathit{exp}_t & \text{List Construction} \\
 & \mid & \mathit{exp}\ \texttt{.}\ \mathit{integer} & \text{Tuple Selection} \\
 & \mid & \texttt{(}\ \mathit{exp}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{exp}_n\ \texttt{)} & \text{Tuple Construction} \\
 & \mid & \mathit{id}\ \texttt{(}\ \mathit{exp}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{exp}_n\ \texttt{)} & \text{Function Application} \\
 & \mid & \mathit{exp}_1\ \mathit{binop}\ \mathit{exp}_2 & \text{Binary Expression} \\
 & \mid & \mathit{unop}\ \mathit{exp} & \text{Unary Expression} \\
 & \mid & \mathit{const} & \text{Constant} \\
 & \mid & \mathit{id} & \text{Variable} \\
 & & & \\
\mathit{patterns} & \rightarrow & \texttt{[ ] =>}\ \mathit{exp}_n\ \texttt{;}\ \mathit{id}_h\ \texttt{:}\ \mathit{id}_t\ \texttt{=>}\ \mathit{exp}_c & \text{List Patterns} \\
 & & & \\
\mathit{const} & \rightarrow & \mathit{integer} & \text{Integer} \\
 & \mid & \texttt{True} \mid \texttt{False} & \text{Boolean} \\
 & \mid & \texttt{[ ]} & \text{Nil} \\
 & \mid & \texttt{Error}\ \texttt{(}\ \mathit{string}\ \texttt{)} & \text{Error} \\
 & & & \\
\mathit{binop} & \rightarrow & \texttt{+} \mid \texttt{*} \mid \texttt{-} \mid \texttt{/} \mid \texttt{\%} & \text{Integer Arithmetic} \\
 & \mid & \texttt{==} \mid \texttt{/=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} & \text{Integer Comparison} \\
 & \mid & \texttt{\&\&} \mid \texttt{||} & \text{Boolean Arithmetic} \\
 & & & \\
\mathit{unop} & \rightarrow & \texttt{-} & \text{Integer Negation} \\
 & \mid & \texttt{\textasciicircum} & \text{Boolean Negation} \\
 & & & \\
\mathit{cdefs} & \rightarrow & \mathit{id}_1^{T_1}\ \texttt{=}\ \mathit{exp}_1 & \text{Constant Definitions} \\
 & & \ldots & (n \geq 0) \\
 & & \mathit{id}_n^{T_n}\ \texttt{=}\ \mathit{exp}_n & \\
 & & & \\
\mathit{fdefs} & \rightarrow & \mathit{id}_1^{T_1}\ \texttt{(}\ \mathit{id}_{11}^{T_{11}}\ \texttt{,}\ \ldots\texttt{,}\ \mathit{id}_{1n_1}^{T_{1n_1}}\ \texttt{)}\ \texttt{=}\ \mathit{exp}_1 & \text{Function Definitions} \\
 & & \ldots & (m \geq 0, \forall 1 \leq i \leq m : n_i > 0) \\
 & & \mathit{id}_m^{T_m}\ \texttt{(}\ \mathit{id}_{m1}^{T_{m1}}\ \texttt{,}\ \ldots\texttt{,}\ \mathit{id}_{mn_m}^{T_{mn_m}}\ \texttt{)}\ \texttt{=}\ \mathit{exp}_m & \\
\end{array}
$$

Figure 4.1: Abstract Syntax

The core therefore represents the first-order monomorphic subset of the language where functions only interact by their arguments. In Section 4.3, we will describe the translation of the full language into this form.

This section does not deal with the semantics of composition expressions (treated in Section 4.2) and of the external program interface (described in Section 4.5).

4.1.2 Notation

In the forthcoming denotational definitions, we will use two special constants:

$$\begin{array}{lcl} \perp & = & \text{“non-termination”} \\ \underline{\perp} & = & \text{“error”} \end{array}$$

$\perp$ denotes the non-termination of a program, while $\underline{\perp}$ denotes some program error. It is common practice to use $\perp$ for both purposes as well, but we feel that the significant operational difference between a program that does not terminate and a program that triggers an error also deserves a denotational distinction. However we do not distinguish between runtime errors and compile-time errors, since the border is vague and depends on the cleverness of the compiler rather than on the properties of the language.

We will denote by $A_{\underline{\perp}}$ the *doubly lifted domain* $(A_\perp)_{\underline{\perp}}$ with the domain ordering $\underline{\perp} \sqsubset \perp \sqsubset a$ $(\forall a \in A)$. *Fixpoint iteration* over a lifted domain is denoted by the operator “fix” and uses $\perp$ (not $\underline{\perp}$!) as its bottom value.

Abstractions constructed by the lambda operator $\underline{\lambda}$ are always *strict* in their argument, i.e. $(\underline{\lambda}x.e)\perp = \perp$ and $(\underline{\lambda}x.e)\underline{\perp} = \underline{\perp}$.

Furthermore, we define the *choice operator* $\rightarrow$ as

$$e_c \rightarrow e_t; e_f \;:=\; \begin{cases} e_t & \text{if } e_c = \text{ true} \\ e_f & \text{if } e_c = \text{ false} \\ \perp & \text{if } e_c = \perp \\ \underline{\perp} & \text{otherwise} \end{cases}$$

Please note that $\rightarrow$ is strict in its first argument (in particular, $(\underline{\perp} \rightarrow e_t; e_f) = \underline{\perp}$) but *not* in both other arguments, e.g. $(\text{true} \rightarrow e_t; \underline{\perp}) = e_t$. We will also frequently use the shortcut $e_c \rightarrow e_t$ for $e_c \rightarrow e_t; \underline{\perp}$ as sort of a *guard condition* asserting that e_c holds.

For any *disjoint union domain* $A + B = \{(1,a) | a \in A\} \cup \{(2,b) | b \in B)\}$, we define the construction operators

$$\begin{array}{l} \text{mkA} : A \rightarrow A + B,\ a \mapsto (1,a) \\ \text{mkB} : B \rightarrow A + B,\ b \mapsto (2,b) \end{array}$$

and the *case operator*

$$\begin{array}{l} \text{case } e_c \text{ of} \\ \quad \text{isA}(x) \rightarrow e_1; \\ \quad \text{isB}(y) \rightarrow e_2 \end{array} \;:=\; \begin{cases} e_1[x \leftarrow a] & \text{if } e_c = (1,a) \text{ for some } a \\ e_2[y \leftarrow b] & \text{if } e_c = (2,b) \text{ for some } b \\ \perp & \text{if } e_c = \perp \\ \underline{\perp} & \text{otherwise} \end{cases}$$

The injection mk$A(\ldots)$ that embeds type A into some union type is always *strict* in its argument i.e. mk$A(\perp) = \perp$ (and likewise for mk$B(\ldots)$).

These notions naturally generalize to an arbitrary number of component domains. The shortcut “case e_c of isA$(a) \rightarrow e_1$” stands for “case e_c of isA$(a) \rightarrow e_1$; isB$(b) \rightarrow \underline{\perp}$”. Please note that the case operator is only strict in its first argument.

4.1.3 Semantic Algebras

In the following subsections, we will define several valuation functions:

P :	Program → Value	Program Value
E :	Expression → ValueEnv → Value	Expression Value
T :	Expression → TypeEnv → Type	Expression Type
DE :	Definitions → ValueEnv → ValueEnv	Value Environment
DT :	Definitions → TypeEnv → TypeEnv	Type Environment

For each expression $[\![exp]\!]$, we will define a rule $\mathbf{T}[\![exp]\!]e$ denoting the *(static) type* of $[\![exp]\!]$ and a rule $\mathbf{E}[\![exp]\!]$ denoting the *(dynamic) value* of $[\![exp]\!]$[1].

The domain "Value" is defined as follows:

Bool =	$\{\text{false}, \text{true}\}$	Boolean
Int =	$\{\text{minint}, \ldots, 0, \ldots, \text{maxint}\}$	Integer
Nil =	*Unit*	Empty List
List =	Value × Value	Non-Empty List
Tuple_i =	Value^i	i-Tuple $(i \geq 2)$
Function_i =	$\text{Value}^0 \rightarrow \ldots \rightarrow \text{Value}^i$	i-ary Function $(i \geq 1)$
Value =	(Bool + Int + Nil + List + Tuple_i $(\forall i \geq 2)$+ Function_i $(\forall i \geq 1))_\perp$	Denotable Value

The usual logical and arithmetic operations are defined on the domains "Bool' and "Int". The domain *Unit* consists of a single element () (and thus does not carry any information contents). Please note that "Value" is a lifted domain that as well contains "error" as "non-termination".

The domain "Type" is defined by

ErrorType =	*Unit*	Error Value
BoolType =	*Unit*	Boolean
IntType =	*Unit*	Integer
NilType =	*Unit*	Empty List
ListType =	Type	Non-Empty List
TupleType_i =	Type^i	i-Tuples $(i \geq 2)$
FunctionType_i =	Type^{i+1}	i-ary Function $(i \geq 1)$
Type =	(ErrorType + BoolType + IntType + NilType + ListType + TupleType_i $(\forall i \geq 2)$ + FunctionType_i $(\forall i \geq 1))_\perp$	Denotable Type

"Type" is a lifted domain that contains $\perp$ to denote "typing error" ($\perp$ is not used). There is no domain corresponding to user-named types, since we assume that these type names have been already replaced by their definition. In the following subsection, we will define several functions operating on "Type".

The environment domains "ValueEnv" and "TypeEnv" are constructed as

ValueEnv =	$(\text{Identifier} \rightarrow \text{Value})_\perp$	Value Environment
TypeEnv =	$(\text{Identifier} \rightarrow \text{Type})_\perp$	Type Environment

[1]The terms "static" and "dynamic" refer to the fact, that the definition of **T** copes without fixpoint iteration, while the definition of **E** does not.

We use the symbol $[\,]$ to denote an empty environment that maps every identifier to $\perp$. The environment $[[\![id]\!] \leftarrow x]$ maps $[\![id]\!]$ to x and every other identifier to $\perp$. $[[\![id]\!] \leftarrow x]\ e$ extends the environment e by a new binding for $[\![id]\!]$ respectively replaces any existing binding for $[\![id]\!]$. Please note that all binding operations are *strict* in their (type respectively value) argument e.g. $[[\![id]\!] \leftarrow \perp]\ e = \perp$.

The following valuation functions will be not refined any more:

I :	Numeral $\rightarrow$ Integer	Integers
FV :	Expression $\rightarrow$ IdentSet	Free Variables
FF :	Expression $\rightarrow$ IdentSet	Free Functions

$\mathbf{I}[\![integer]\!]$ translates the textual description of an integer number into its *numerical value*. The result of the function is $\perp$ if this value is outside the range $\{minint, \ldots, maxint\}$.

$\mathbf{FV}[\![exp]\!]$ is the set of *free variables* in exp i.e. the identifiers of those variables that are referenced but not defined within exp. Likewise, $\mathbf{FF}[\![exp]\!]$ denotes the set of *free functions* in exp i.e. the identifiers of those functions that are applied but not defined within exp.

4.1.4 Typing

We assume that user-defined type names have been replaced by their definitions and that polymorphic functions have been replaced by their monomorphic counterparts. Thus on the level of the core language types are constructed from a small set of type constants and constructors.

However even on the core level "type compatibility" still does not mean "type equality". The reason is that two of the types need special consideration:

- "mkErrorType()" is compatible with any type.
- "mkNilType()" is compatible with any list type.

These types are not available at the user-level of the language but arise from the polymorphic constants `Error` and `[ ]` without which programming would be rather cumbersome.

The following function defines when two types are considered compatible by the denotation function **T**:

$$
\begin{array}{l}
\text{eqType} : \text{Type} \times \text{Type} \rightarrow \text{Bool} \\
\text{eqType}(T_1, T_2) = \\
\quad T_1 = \text{ mkErrorType}() \vee T_2 = \text{ mkErrorType}() \rightarrow \text{ true;} \\
\quad \text{case } T_1 \text{ of} \\
\quad\quad \text{isIntType}() \rightarrow T_2 = \text{ mkIntType}() \\
\quad\quad \text{isBoolType}() \rightarrow T_2 = \text{ mkBoolType}() \\
\quad\quad \text{isTupleType}_n(t_1, \ldots, t_n) \rightarrow \\
\quad\quad\quad \text{case } T_2 \text{ of} \\
\quad\quad\quad\quad \text{isTupleType}_m(s_1, \ldots, s_m) \rightarrow \\
\quad\quad\quad\quad\quad m = n \wedge \forall 1 \leq i \leq m :\ \text{eqType}(t_i, s_i) \\
\quad\quad\quad\quad \text{otherwise } \rightarrow \text{ false} \\
\quad\quad \text{isNilType}() \rightarrow \\
\quad\quad\quad \text{case } T_2 \text{ of} \\
\quad\quad\quad\quad \text{isNilType}() \rightarrow \text{ true} \\
\quad\quad\quad\quad \text{isListType}(t) \rightarrow \text{ true} \\
\quad\quad\quad\quad \text{otherwise } \rightarrow \text{ false}
\end{array}
$$

$\text{isListType}(t_1) \rightarrow$
  $\text{case } T_2 \text{ of}$
    $\text{isNilType}() \rightarrow \text{ true}$
    $\text{isListType}(t_2) \rightarrow \text{ eqType}(t_1, t_2)$
    $\text{otherwise } \rightarrow \text{ false}$

The following function returns the more defined of two compatible types. This is required to determine the type of "multi-branched" expressions such as conditional or case expressions.

$\text{joinType} : \text{Type} \times \text{Type} \rightarrow \text{Type}$
$\text{joinType}(T_1, T_2) =$
  $T_1 = \text{ mkErrorType}() \rightarrow T_2;$
  $T_2 = \text{ mkErrorType}() \rightarrow T_1;$
  $\text{case } T_1 \text{ of}$
    $\text{isIntType}() \rightarrow (T_2 = \text{ mkIntType}() \rightarrow T_1)$
    $\text{isBoolType}() \rightarrow (T_2 = \text{ mkBoolType}() \rightarrow T_1)$
    $\text{isTupleType}_n(t_1, \ldots, t_n) \rightarrow$
      $\text{case } T_2 \text{ of isTupleType}_m(s_1, \ldots, s_m) \rightarrow$
        $m = n \rightarrow \text{ mkTupleType}(\text{joinType}(t_1, s_1), \ldots, \text{joinType}(t_n, s_n))$
    $\text{isNilType}() \rightarrow$
      $\text{case } T_2 \text{ of}$
        $\text{isNilType}() \rightarrow \text{ mkNilType}()$
        $\text{isListType}(t) \rightarrow T_2$
    $\text{isListType}(t_1) \rightarrow$
      $\text{case } T_2 \text{ of}$
        $\text{isNilType}() \rightarrow T_1$
        $\text{isListType}(t_2) \rightarrow \text{ mkListType}(\text{joinType}(t_1, t_2))$

4.1.5 Program

$$\mathbf{P}[\![\texttt{let } \mathit{fdefs} \texttt{ in } \mathit{exp}]\!] = \mathbf{E}[\![\texttt{let } \mathit{fdefs} \texttt{ in } \mathit{exp}]\!][\,]$$

A program is an expression that is evaluated in an empty environment. The denotation function **E** takes as an additional argument the current environment defining the value of each global constant and the definition of each function. **E** is *strict* in this argument since it is defined by rules of the form

$$\mathbf{E}[\![\mathit{exp}]\!] = \underline{\lambda} e. \ldots$$

(for every kind of expression *exp*) using the strict lambda operator $\underline{\lambda}$. Furthermore, since an environment is strict in its individual bindings, the denotation of an expression will be $\underline{\perp}$ respectively $\perp$ whenever any identifier in its environment is bound to $\underline{\perp}$ respectively $\perp$.

In the following subsections, we will give for each kind of expression a rule for the definition of **E** (the value denotation) and a separate rule for the definition of **T** (the type denotation). The definition of **T** takes an environment as an additional argument that gives the type of each global constant or function.

Each rule of the form $\mathbf{E}[\![\mathit{exp}]\!] = \underline{\lambda} e.v$ has then to be interpreted as

$$\mathbf{E}[\![\mathit{exp}]\!] = \underline{\lambda} e.\underline{\lambda} t.\mathbf{T}[\![\mathit{exp}]\!] t \neq \underline{\perp} \rightarrow v$$

i.e. only well-typed programs have a valid denotation.

4.1.6 Function Definitions

$$\mathbf{T}[\![\texttt{let } \mathit{fdefs} \texttt{ in } \mathit{exp}]\!] = \underline{\lambda} e.\mathbf{T}[\![\mathit{exp}]\!](\mathbf{DT}[\![\mathit{fdefs}]\!]e)$$
$$\mathbf{E}[\![\texttt{let } \mathit{fdefs} \texttt{ in } \mathit{exp}]\!] = \underline{\lambda} e.\mathbf{E}[\![\mathit{exp}]\!](\mathbf{DE}[\![\mathit{fdefs}]\!]e)$$

The type of the return value of a function must match the declared return type:

$$\begin{array}{l}
\mathbf{DT}[\![\mathit{id}_1^{T_1}\texttt{(}\mathit{id}_{11}^{T_{11}}\texttt{,}\ldots\texttt{,}\mathit{id}_{1n_1}^{T_{1n_1}}\texttt{)} \texttt{ = } \mathit{exp}_1 \ \ldots\ \mathit{id}_m^{T_m}\texttt{(}\mathit{id}_{m1}^{T_{m1}}\texttt{,}\ldots\texttt{,}\mathit{id}_{mn_m}^{T_{mn_m}}\texttt{)} \texttt{ = } \mathit{exp}_m]\!] = \underline{\lambda} e. \\
\quad (\forall 1 \le i, j \le m : i = j \vee [\![\mathit{id}_i]\!] \neq [\![\mathit{id}_j]\!]) \rightarrow \\
\quad\quad \text{let } e' = \\
\quad\quad\quad [[\![\mathit{id}_1]\!] \leftarrow \ \text{mkFunctionType}(T_1, T_{11}, \ldots, T_{1n_1}) \\
\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad [\![\mathit{id}_m]\!] \leftarrow \ \text{mkFunctionType}(T_m, T_{m1}, \ldots, T_{mn_m})]\ e \\
\quad\quad \text{in} \\
\quad\quad\quad (\text{eqType}(T_1, \mathbf{T}[\![\mathit{exp}_1]\!][[\![\mathit{id}_{11}]\!] \leftarrow T_{11} \ldots [\![\mathit{id}_{1n_1}]\!] \leftarrow T_{1n_1}]\ e') \\
\quad\quad\quad \wedge\ \ldots\ \wedge \\
\quad\quad\quad \text{eqType}(T_m, \mathbf{T}[\![\mathit{exp}_m]\!][[\![\mathit{id}_{m1}]\!] \leftarrow T_{m1} \ldots [\![\mathit{id}_{mn_m}]\!] \leftarrow T_{mn_m}]\ e')) \rightarrow e'
\end{array}$$

Within a block, the order of function definitions does not matter. Each occurrence of the name of a function defined in this block refers to this particular definition (even if the reference precedes the definition and a function of the same name has been defined in the outer environment). Consequently, function definitions may be (mutually) recursive. It is not allowed to define the same name twice within a block.

$$\begin{array}{l}
\mathbf{DE}[\![\mathit{id}_1^{T_1}\texttt{(}\mathit{id}_{11}^{T_{11}}\texttt{,}\ldots\texttt{,}\mathit{id}_{1n_1}^{T_{1n_1}}\texttt{)} \texttt{ = } \mathit{exp}_1 \ \ldots\ \mathit{id}_m^{T_m}\texttt{(}\mathit{id}_{m1}^{T_{m1}}\texttt{,}\ldots\texttt{,}\mathit{id}_{mn_m}^{T_{mn_m}}\texttt{)} \texttt{ = } \mathit{exp}_m]\!] = \underline{\lambda} e. \\
\quad \text{fix}(\lambda e'. \\
\quad\quad [[\![\mathit{id}_1]\!] \leftarrow \ \text{mkFunction}_{n_1}(\underline{\lambda} v_1 \ldots \underline{\lambda} v_{n_1}. \\
\quad\quad\quad \mathbf{E}[\![\mathit{exp}_1]\!][[\![\mathit{id}_{11}]\!] \leftarrow v_1 \ldots [\![\mathit{id}_{1n_1}]\!] \leftarrow v_{n_1}]\ e') \\
\quad\quad \ldots \\
\quad\quad [\![\mathit{id}_m]\!] \leftarrow \ \text{mkFunction}_{n_m}(\underline{\lambda} v_1 \ldots \underline{\lambda} v_{n_m}. \\
\quad\quad\quad \mathbf{E}[\![\mathit{exp}_m]\!][[\![\mathit{id}_{m1}]\!] \leftarrow v_1 \ldots [\![\mathit{id}_{mn_m}]\!] \leftarrow v_{n_m}]\ e')]\ e)
\end{array}$$

Functions have at least one argument; instead of a zero-argument function, a constant must be defined.

4.1.7 Constant Definitions

$$\mathbf{T}[\![\texttt{let } \mathit{cdefs} \texttt{ in } \mathit{exp}]\!] = \underline{\lambda} e.\mathbf{T}[\![\mathit{exp}]\!](\mathbf{DT}[\![\mathit{cdefs}]\!]e)$$
$$\mathbf{E}[\![\texttt{let } \mathit{cdefs} \texttt{ in } \mathit{exp}]\!] = \underline{\lambda} e.\mathbf{E}[\![\mathit{exp}]\!](\mathbf{DE}[\![\mathit{cdefs}]\!]e)$$

The type of a constant value must match the declared type. In the full language, it is actually not required to explicitly type constants; the type of a constant is just the type of its value. However, by explicit type annotations, it is possible to give a constant name a type that is more specific than the type of its value (in particular a concrete list type for a constant defined to be the empty list).

$$\begin{array}{l}
\mathbf{DT}[\![\mathit{id}_1^{T_1} \texttt{ = } \mathit{exp}_1 \texttt{ ; } \ldots \texttt{ ; } \mathit{id}_n^{T_n} \texttt{ = } \mathit{exp}_n]\!] = \underline{\lambda} e. \\
\quad (\forall 1 \le i, j \le n : i = j \vee [\![\mathit{id}_i]\!] \neq [\![\mathit{id}_j]\!]) \rightarrow \\
\quad\quad \text{let } e_1 = \ \text{eqType}(T_{p_1}, \mathbf{T}[\![\mathit{exp}_{p_1}]\!]e) \rightarrow [[\![\mathit{id}_{p_1}]\!] \leftarrow T_{p_1}]\ e_0 \text{ in} \\
\quad\quad \text{let } e_2 = \ \text{eqType}(T_{p_2}, \mathbf{T}[\![\mathit{exp}_{p_2}]\!]e_1) \rightarrow [[\![\mathit{id}_{p_2}]\!] \leftarrow T_{p_2}]\ e_1 \text{ in} \\
\quad\quad \ldots \\
\quad\quad \text{let } e_n = \ \text{eqType}(T_{p_n}, \mathbf{T}[\![\mathit{exp}_{p_n}]\!]e_{n-1}) \rightarrow [[\![\mathit{id}_{p_n}]\!] \leftarrow T_{p_n}]\ e_{n-1} \text{ in} \\
\quad\quad\quad e_n
\end{array}$$

As for function definitions, within a block the order of constant definitions does not matter. Each occurrence of a constant name defined in the block refers to this particular definition. However, in contrast to function definitions, constant definitions must *not* be recursive or mutually recursive. There must exist some reordering of constant definitions in which each reference to a constant is preceded by its definition.

$$\begin{array}{l}
\mathbf{DE}[\![id_1^{T_1} \texttt{ = } exp_1 \texttt{ ; } \ldots \texttt{ ; } id_n^{T_n} \texttt{ = } exp_n]\!] = \underline{\lambda} e. \\
\quad \text{let } e_1 = [[\![id_{p_1}]\!] \leftarrow \mathbf{E}[\![exp_{p_1}]\!] e_0]\ e \text{ in} \\
\quad \text{let } e_2 = [[\![id_{p_2}]\!] \leftarrow \mathbf{E}[\![exp_{p_2}]\!] e_1]\ e_1 \text{ in} \\
\quad \ldots \\
\quad \text{let } e_n = [[\![id_{p_n}]\!] \leftarrow \mathbf{E}[\![exp_{p_n}]\!] e_{n-1}]\ e_{n-1} \text{ in} \\
\qquad e_n
\end{array}$$

$$p = \text{topSort}(\{1 \ldots n\}, \{(i,j) | i,j \in \ \{1 \ldots n\}, [\![id_j]\!] \in \mathbf{FV}[\![exp_i]\!]\})$$

$p = \text{topSort}(V,E)$ is a topological order of the graph (V,E). If such an order does not exist (i.e. the graph contains cycles), the result of topSort (and thus also the environment returned by **DT** respectively **DE**) is $\underline{\perp}$.

4.1.8 Conditional

$$\begin{array}{l}
\mathbf{T}[\![\texttt{if } exp_c \texttt{ then } exp_t \texttt{ else } exp_f]\!] = \underline{\lambda} e. \\
\quad \text{eqType}(\mathbf{T}[\![exp_c]\!] e,\ \text{mkBoolType}()) \rightarrow \\
\qquad \text{joinType}(\mathbf{T}[\![exp_t]\!] e, \mathbf{T}[\![exp_f]\!] e) \\
\mathbf{E}[\![\texttt{if } exp_c \texttt{ then } exp_t \texttt{ else } exp_f]\!] = \underline{\lambda} e. \\
\quad \text{case } (\mathbf{E}[\![exp_c]\!] e) \text{ of isBool}(v) \rightarrow \\
\qquad v \rightarrow \mathbf{E}[\![exp_t]\!] e ; \mathbf{E}[\![exp_f]\!] e
\end{array}$$

The conditional construct is one of the few operations in the language that is non-strict in some of its arguments (namely the the branch expressions). Consequently, it may encapsulate in one branch a non-terminating or erroneous subcomputation; if the other branch is selected and returns a valid result, the overall result is also valid. However, the types of both branches must be compatible, even if the selection expression always determines the same branch.

4.1.9 List Selection

$$\begin{array}{l}
\mathbf{T}[\![\texttt{case } exp_s \texttt{ of [ ] => } exp_n \texttt{ ; } id_h \texttt{:} id_t \texttt{ => } exp_c]\!] = \underline{\lambda} e. \\
\quad \text{case } \mathbf{T}[\![exp_s]\!] e \text{ of} \\
\qquad \text{isErrorType}() \rightarrow \text{joinType}(\mathbf{T}[\![exp_n]\!] e, \mathbf{T}[\![exp_c]\!] e); \\
\qquad \text{isNilType}() \rightarrow \text{joinType}(\mathbf{T}[\![exp_n]\!] e, \mathbf{T}[\![exp_c]\!] e); \\
\qquad \text{isListType}(T) \ \rightarrow \\
\qquad\quad \text{joinType}(\mathbf{T}[\![exp_n]\!] e, \mathbf{T}[\![exp_c]\!] [[\![id_h]\!] \leftarrow T, [\![id_t]\!] \leftarrow\ \text{mkListType}(T)]\ e
\end{array}$$

There is a slight problem in typing the `case` construct if the selection expression is the constant `[ ]` or `Error`. In this pathological case we are not able to derive types for the head and tail identifiers of the second branch; these identifiers must not be used then.

$$\begin{array}{l}
\mathbf{E}[\![\texttt{case } exp_s \texttt{ of [ ] => } exp_n \texttt{ ; } id_h \texttt{:} id_t \texttt{ => } exp_c]\!] = \underline{\lambda} e. \\
\quad \text{case } (\mathbf{E}[\![exp_s]\!] e) \text{ of} \\
\qquad \text{isNil}() \rightarrow \mathbf{E}[\![exp_n]\!] e; \\
\qquad \text{isList}(h,t) \rightarrow\ \mathbf{E}[\![exp_c]\!] [[\![id_h]\!] \leftarrow h, [\![id_t]\!] \leftarrow t]\ e
\end{array}$$

Like the conditional, the case expression is only strict in its first argument and may encapsulate a non-terminating or erroneous branch expression.

4.1.10 List Construction

$$
\begin{array}{l}
\mathbf{T}[\![exp_h \ \texttt{:}\ exp_t]\!] = \underline{\lambda} e. \\
\quad \text{case } (\mathbf{T}[\![exp_t]\!] e) \text{ of} \\
\qquad \text{isErrorType}() \rightarrow \text{mkListType}(\mathbf{T}[\![exp_h]\!] e) \\
\qquad \text{isNilType}() \rightarrow \text{mkListType}(\mathbf{T}[\![exp_h]\!] e) \\
\qquad \text{isListType}(t) \rightarrow \text{joinType}(t, \mathbf{T}[\![exp_h]\!] e) \\
\mathbf{E}[\![exp_h \ \texttt{:}\ exp_t]\!] = \underline{\lambda} e. \\
\quad \text{mkList}(\mathbf{E}[\![exp_h]\!] e, \mathbf{E}[\![exp_t]\!] e)
\end{array}
$$

In constructor expressions, the type of the list head must match the type of the list tail (unless the tail is `[ ]` or `Error`).

4.1.11 Tuple Selection

$$
\begin{array}{l}
\mathbf{T}[\![exp \ \texttt{.}\ integer]\!] = \underline{\lambda} e. \\
\quad \text{case } (\mathbf{T}[\![exp]\!] e) \text{ of} \\
\qquad \text{isErrorType}() \rightarrow \text{isErrorType}() \\
\qquad \text{isTupleType}_n(t_1, \ldots, t_n) \rightarrow \\
\qquad\quad \text{let } i = \mathbf{I}[\![integer]\!] \\
\qquad\qquad \text{in } 1 \le i \le n \rightarrow t_i \\
\mathbf{E}[\![exp \ \texttt{.}\ integer]\!] = \underline{\lambda} e. \\
\quad \text{case } (\mathbf{E}[\![exp]\!] e) \text{ of isTuple}_n(v_1, \ldots, v_n) \rightarrow \\
\qquad \text{let } i = \mathbf{I}[\![integer]\!] \\
\qquad \text{in } 1 \le i \le n \rightarrow v_i
\end{array}
$$

This form of the selection of a tuple field is actually not available in the full language. It is a compiler-internal translation of the user-level matching construct `(`a_1`, ..., `a_n`) =` t where the a_i become the fields of the tuple t.

4.1.12 Tuple Construction

$$
\begin{array}{l}
\mathbf{T}[\![\texttt{(} exp_1 \texttt{,} \ \ldots \ \texttt{,} \ exp_n \texttt{)}]\!] = \underline{\lambda} e. \\
\quad \text{mkTupleType}_n(\mathbf{T}[\![exp_1]\!] e, \ldots, \mathbf{T}[\![exp_n]\!] e) \\
\mathbf{E}[\![\texttt{(} exp_1 \texttt{,} \ \ldots \ \texttt{,} \ exp_n \texttt{)}]\!] = \underline{\lambda} e. \\
\quad \text{mkTuple}_n(\mathbf{E}[\![exp_1]\!] e, \ldots, \mathbf{E}[\![exp_n]\!] e)
\end{array}
$$

In the full language, a single field tuple `(`t`)` is identified with its content t as well as a single field tuple type with its base type. Hence, tuples in the core language have at least two fields.

4.1.13 Function Application

$$
\begin{array}{l}
\mathbf{T}[\![id \texttt{(} \ exp_1 \texttt{,} \ \ldots \ \texttt{,} \ exp_n \texttt{)}]\!] = \underline{\lambda} e. \\
\quad \text{case } e([\![id]\!]) \text{ of isFunctionType}_m(t_r, t_1, \ldots t_m) \rightarrow \\
\qquad (m = n \wedge \forall 1 \le i \le n : \ \text{eqType}(t_i, \mathbf{T}[\![exp_i]\!] e)) \rightarrow t_r
\end{array}
$$

$$
\begin{array}{l}
\mathbf{E}[\![id(\texttt{ } exp_1 \texttt{,} \ \ldots \ \texttt{,} \ exp_n \texttt{)}]\!] = \underline{\lambda} e. \\
\quad \text{case } e([\![id]\!]) \text{ of isFunction}(f) \rightarrow \\
\quad\quad f \mathbf{E}[\![exp_1]\!] e \ \ldots \ \mathbf{E}[\![exp_n]\!] e
\end{array}
$$

Functions have at least one argument, zero-argument functions are subsumed by constants.

4.1.14 Binary Expressions

On integers a variety of binary operations are defined. We only give the semantics of two examples; the remaining definitions are similar. Please note that, although the operation symbols themselves determine the result type, the result type is only defined if also the argument types are correct. By this we force **T** to reject constructions like `1 + if True then 2 else [ ]` that are allowed by **E**.

$$
\begin{array}{l}
\mathbf{T}[\![exp_1 \texttt{ + } exp_2]\!] = \underline{\lambda} e. \\
\quad \text{eqType}(\mathbf{T}[\![exp_1]\!] e, \ \text{mkIntType}()) \rightarrow \\
\quad \text{eqType}(\mathbf{T}[\![exp_2]\!] e, \ \text{mkIntType}()) \rightarrow \\
\quad\quad \text{mkIntType}() \\
\mathbf{E}[\![exp_1 \texttt{ + } exp_2]\!] = \underline{\lambda} e. \\
\quad \text{case } \mathbf{E}[\![exp_1]\!] e \text{ of isInt}(i_1) \rightarrow \\
\quad \text{case } \mathbf{E}[\![exp_2]\!] e \text{ of isInt}(i_2) \rightarrow \\
\quad\quad \text{mkInt}(i_1 + i_2)
\end{array}
$$

$$
\begin{array}{l}
\mathbf{T}[\![exp_1 \texttt{ == } exp_2]\!] = \underline{\lambda} e. \\
\quad \text{eqType}(\mathbf{T}[\![exp_1]\!] e, \ \text{mkIntType}()) \rightarrow \\
\quad \text{eqType}(\mathbf{T}[\![exp_2]\!] e, \ \text{mkIntType}()) \rightarrow \\
\quad\quad \text{mkBoolType}() \\
\mathbf{E}[\![exp_1 \texttt{ == } exp_2]\!] = \underline{\lambda} e. \\
\quad \text{case } \mathbf{E}[\![exp_1]\!] e \text{ of isInt}(i_1) \rightarrow \\
\quad \text{case } \mathbf{E}[\![exp_2]\!] e \text{ of isInt}(i_2) \rightarrow \\
\quad\quad \text{mkBool}(i_1 = i_2)
\end{array}
$$

Logical conjunction and disjunction are only strict in their first argument:

$$
\begin{array}{l}
\mathbf{T}[\![exp_1 \texttt{ \&\& } exp_2]\!] = \underline{\lambda} e. \\
\quad \text{eqType}(\mathbf{T}[\![exp_1]\!] e, \ \text{mkBoolType}()) \rightarrow \\
\quad \text{eqType}(\mathbf{T}[\![exp_2]\!] e, \ \text{mkBoolType}()) \ \rightarrow \\
\quad\quad \text{mkBoolType}() \\
\mathbf{E}[\![exp_1 \texttt{ \&\& } exp_2]\!] = \underline{\lambda} e. \\
\quad \text{case } \mathbf{E}[\![exp_1]\!] e \text{ of isBool}(b_1) \rightarrow b_1 \rightarrow \\
\quad\quad \text{case } \mathbf{E}[\![exp_2]\!] e \text{ of isBool}(b_2) \rightarrow \text{mkBool}(b_2); \\
\quad\quad \text{mkBool(false)}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{T}[\![exp_1 \texttt{ || } exp_2]\!] = \underline{\lambda} e. \\
\quad \text{eqType}(\mathbf{T}[\![exp_1]\!] e, \ \text{mkBoolType}()) \rightarrow \\
\quad \text{eqType}(\mathbf{T}[\![exp_2]\!] e, \ \text{mkBoolType}()) \ \rightarrow \\
\quad\quad \text{mkBoolType}() \\
\mathbf{E}[\![exp_1 \texttt{ || } exp_2]\!] = \underline{\lambda} e. \\
\quad \text{case } \mathbf{E}[\![exp_1]\!] e \text{ of isBool}(b_1) \rightarrow b_1 \rightarrow \\
\quad\quad \text{mkBool(true)}; \\
\quad\quad \text{case } \mathbf{E}[\![exp_2]\!] e \text{ of isBool}(b_2) \rightarrow \text{mkBool}(b_2)
\end{array}
$$

Actually we have a tight correspondence between the logical operators and the conditional construct, e.g. $\mathbf{E}[\![exp_1 \texttt{ \&\& } exp_2]\!] = \mathbf{E}[\![\texttt{if } exp_1 \texttt{ then } exp_2 \texttt{ else False}]\!]$. However, we keep

exp	$\rightarrow$	*composer iterator*	Composition
composer	$\rightarrow$	`comp (` *id* `,` *exp* `)`	Composer
	$\mid$	`rcomp (` *id* `,` *exp* `)`	
iterator	$\rightarrow$	`[` *exp* `|` $iter_1$ `|` ... `|` $iter_n$ `]`	Iterator List
iter	$\rightarrow$	pre_1 `,` ... `,` pre_n `;` *exp* `;` $post_1$ `,` ... `,` $post_m$	Iterator
pre	$\rightarrow$	id_h `:` id_t `<-` *exp*	List Iteration
	$\mid$	*id* `<-` exp_s `..` exp_e `+=` exp_i	Upward Iteration
	$\mid$	*id* `<-` exp_s `..` exp_e `-=` exp_i	Downward Iteration
	$\mid$	*id* `=` *exp*	General Iteration
	$\mid$	*def*	Local Definition
post	$\rightarrow$	*id* `' =` *exp*	General Iteration
	$\mid$	*def*	Local Definition

Figure 4.2: Iterated Composition Syntax

them distinct since the compiler applies different optimizations and generates different target code for both constructs.

4.1.15 Unary Expressions

$$\mathbf{T}[\![\texttt{-}\ exp]\!] = \underline{\lambda}e.\ \text{eqType}(\mathbf{T}[\![exp]\!]e,\ \text{mkIntType}()) \ \rightarrow\ \text{mkIntType}()$$
$$\mathbf{E}[\![\texttt{-}\ exp]\!] = \underline{\lambda}e.\ \text{case } \mathbf{E}[\![exp]\!]e \text{ of isInt}(i) \rightarrow \text{mkInt}(-i)$$

$$\mathbf{T}[\![\texttt{\^{}}\ exp]\!] = \underline{\lambda}e.\ \text{eqType}(\mathbf{T}[\![exp]\!]e,\ \text{mkBoolType}()) \ \rightarrow\ \text{mkBoolType}()$$
$$\mathbf{E}[\![\texttt{\^{}}\ exp]\!] = \underline{\lambda}e.\ \text{case } \mathbf{E}[\![exp]\!]e \text{ of isBool}(b) \rightarrow \text{mkBool}(\neg b)$$

For type checking, the same remarks apply as in the case of binary expressions.

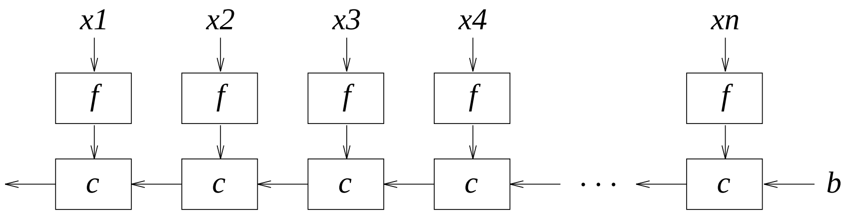
4.1.16 Constants

$$\mathbf{T}[\![integer]\!] = \underline{\lambda}e.\ \text{mkIntType}()$$
$$\mathbf{E}[\![integer]\!] = \underline{\lambda}e.\ \text{mkInt}(\mathbf{I}[\![integer]\!])$$

$$\mathbf{T}[\![\texttt{True}]\!] = \underline{\lambda}e.\ \text{mkBoolType}()$$
$$\mathbf{E}[\![\texttt{True}]\!] = \underline{\lambda}e.\ \text{mkBool(true)}$$

$$\mathbf{T}[\![\texttt{False}]\!] = \underline{\lambda}e.\ \text{mkBoolType}()$$
$$\mathbf{E}[\![\texttt{False}]\!] = \underline{\lambda}e.\ \text{mkBool(false)}$$

$$\mathbf{T}[\![\texttt{[ ]}]\!] = \underline{\lambda}e.\ \text{mkNilType}()$$
$$\mathbf{E}[\![\texttt{[ ]}]\!] = \underline{\lambda}e.\ \text{mkNil}()$$



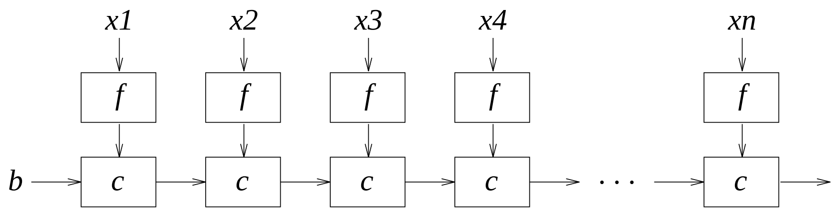


Figure 4.3: Iterated Compositions

$\mathbf{T}[\![$`Error ( `$string$` )`$]\!] = \underline{\lambda} e.$ mkErrorType()
$\mathbf{E}[\![$`Error ( `$string$` )`$]\!] = \underline{\lambda} e. \underline{\perp}$

Please note that while the value denotation of `Error` is $\underline{\perp}$, its type denotation is not. The application of $\underline{\perp}$ in the definition of $\mathbf{T}$ denotes a typing error that is propagated to the top. However, the type of an expression involving `Error` is in general different from "mkErrorType()" e.g. `if x <= 0 then Error ("neg") else x-1` is of type "mkIntType()".

4.1.17 Variables

$\mathbf{T}[\![id]\!] = \underline{\lambda} e.\ e(id)$
$\mathbf{E}[\![id]\!] = \underline{\lambda} e.\ e(id)$

The type respectively value denotation of a variable is $\underline{\perp}$ if the variable is not defined in the current environment.

4.2 Iterated Compositions

4.2.1 Definition

The core language is basically first-order; however, there exist *iterated compositions* as a sort of higher-order constructs to express typical program patterns without applying recursion. An iterated composition consists of two parts (see Figure 4.2):

- The *iterator* generates a sequence of variable bindings. An expression is evaluated in each of these environments yielding a sequence of values.

- The *composer* folds the generated value sequence into a single value. There are several composers that may be parameterized by a user-defined function.

First we will describe how value sequences are folded (see Figure 4.3). Let $\langle a_1, a_2, \ldots, a_n\rangle$ be the sequence of values generated by the iterator part of a iterated composition, f the name of a function and b an expression:

- *Standard Composition:* The composer `comp(`f`,`b`)` computes the value

  $$f(a_1, f(a_2, \ldots, f(a_n, b)))$$

- *Reverse Composition:* The composer `rcomp(`f`,`b`)` computes the value

  $$f(a_n, f(a_{n-1}, \ldots, f(a_1, b), \ldots))$$

`comp` and `compr` essentially correspond to the higher order functions `foldr` (fold right) and `foldl` (fold left) of several functional languages.

The composer function must be of type $f : (A, B) \rightarrow B$ where A is the type of the a_i and B is the type of b which is also the type of the composition result (b, if the value sequence is empty).

There are several other composers that have been predefined in terms of `comp` and `rcomp`:

`list`	=	`comp(:,[ ])`	List Construction
`rlist`	=	`rcomp(:,[ ])`	Reverse List Construction
`loop(`b`)`	=	`rcomp(`$\lambda(x, y).x$`,`b`)`	Loop Iteration

The composer `loop(`b`)` returns the last element of the value sequence.

Now let us turn to the iterator part which consists of an expression and the actual iterator:

`[` *exp* `|` *iterator* `]`

exp is evaluated in each of the bindings generated by *iterator* yielding the value sequence $\langle a_1, a_2, \ldots, a_n\rangle$ to be combined. *iterator* consists of a sequence

$iter_1$ `|` ... `|` $iter_n$

For each environment generated by $iter_1$ `|` ... `|` $iter_{i-1}$, $iter_i$ generates a sequence of environments; the concatenation of all these sequences is the environment sequence of $iter_i$. *exp* is evaluated in the environment sequence of $iter_n$.

Each $iter_i$ may be of one of the following forms:

- *List Iterator:*

  h`:`t `<-` l

  creates a sequence of environments in which h is bound consequently to each element l_i of the list l and t is bound to the corresponding rest list l_{i+1}`:` ... `:`l_n. The qualifier `:`t may be omitted.

- *Arithmetic Iterator:*

  x `<-` a`..`b `+=` c

  creates a sequence of environments in which x is bound consequently to $x_0 = a, x_1 = a + c, x_2 = a + 2 * c, \ldots$ as long as $x_i \leq b$. If the qualifier `+=` c is omitted, $c = 1$ is assumed. If the qualifier `-=` c is used, x is bound to $x_0 = a, x_1 = a - c, x_2 = a - 2*c, \ldots$ as long as $x_i \geq b$.

- *General Iterator:*

  $y = r \; ; \; s_y \; ; \; y' = t_y$

  creates a sequence of environments in which y is bound consequently to $y_0 = r, y_1 = t_{y_0}, y_2 = t_{y_1}, \ldots$ as long as s_{y_i} holds.

- *Superposition:* Iterators may be superposed

  $pre_1 \; , \; \ldots \; , \; pre_n \; ; \; cond \; ; \; post_m \; , \; \ldots \; , \; post_m$

  where each pre_i may be an arithmetic iterator, a list iterator or the initialization part of a general iterator. *cond* is a boolean condition and each $post_j$ is the successor part of a general iterator. This iterator superposition creates a sequence of environments where all iterated variables are bound simultaneously to their corresponding value sequences. The environment sequence is terminated when at least one of the value sequences is terminated.

  cond may refer to all iterated variables, not only those defined by general iterators, as well as the successor expressions in the $post_j$. Furthermore, each of the pre_i or $post_j$ may be a local definition. The name defined in a pre_i definition may be referenced in any definition (*not* iterator) $pre_{i'}$, in *cond* and in all $post_j$. The name defined in a $post_j$ definition may be referenced in any $post_{j'}$ successor expression or definition.

  Furthermore all names defined in a local *pre* or *post* definition of an iterator $iter_i$ may be referenced in all iterators $iter_j$ with $j > i$. Any of the parts *pre*, *cond* and *post* may be empty.

  These rules for the superposition of iterators may look rather complicated but become clear from the translation of composition expressions shown in Subsection 4.2.3.

4.2.2 Examples

Some examples will demonstrate the application of iterated compositions:

```
map(f,l) = list [ f(x) | x <- l ]
```

construct a list by applying f to each element of l.

```
map(f,l) = list [ y | x <- l, y = f(x) ]
```

does the same using a local definition.

```
reverse(l) = rlist [ x | x <- l ]
```

returns l with the element order reverted.

```
fromto(a,b) = list [ x | x <- a..b ]
```

returns the integer sequence $[a \ldots b]$.

```
pow2(n) = list [ x | x = 1; x <= n; x' = x*2 ]
```

constructs the list $[1, 2, 4, 8, \ldots, 2^m]$ where $2^m \leq n < 2^{m+1}$.

```
while(p,l) = list [ x | x <- l; p(x) ]
```

returns the longest initial subsequence of l such that for each element p holds.

```
filter(p,l) = list [ x | x <- l |; p(x) ]
```

$$\mathbf{TC}[\![\texttt{comp}(f,b)\ \texttt{[}\ e\ \texttt{|}\ I\ \texttt{]}]\!] = \mathbf{TI}[\![I]\!][\![f]\!][\![e]\!][\![b]\!]$$

$$\mathbf{TI}[\![\ ]\!][\![f]\!][\![e]\!][\![v]\!] \ = \ [\![f(e,v)]\!]$$

$$\mathbf{TI}[\![h_i\texttt{:}t_i\texttt{<-}l_i\texttt{,}\ x_j\texttt{<-}a_j\texttt{..}b_j\texttt{+=}c_j\texttt{,}\ y_k\texttt{=}u_k\texttt{,}\ \mathit{defs}_1\texttt{;}\ s\texttt{;}\ y_k\texttt{'}\ \texttt{=}\ r_k\texttt{,}\ \mathit{defs}_2\ \texttt{|}\ R]\!][\![f]\!][\![e]\!][\![v]\!] \ =$$

$$\left[\!\!\left[\begin{array}{l}
\texttt{let} \\
\quad F(z_1,\ \ldots,\ z_m,\ x_1,\ \ldots,\ x_n,\ y_1,\ \ldots,\ y_o)\ \texttt{=} \\
\quad\quad \texttt{case}\ z_1\ \texttt{of} \\
\quad\quad\quad \texttt{[ ] =>}\ \bar{v} \\
\quad\quad\quad h_1\texttt{:}t_1\ \texttt{=>} \\
\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad \texttt{case}\ z_m\ \texttt{of} \\
\quad\quad\quad\quad\quad\quad \texttt{[ ] =>}\ \bar{v} \\
\quad\quad\quad\quad\quad\quad h_m\texttt{:}t_m\ \texttt{=>} \\
\quad\quad\quad\quad\quad\quad\quad \texttt{if}\ x_1\ \texttt{>}\ \bar{b}_1\ \texttt{||}\ \ldots\ \texttt{||}\ x_n\ \texttt{>}\ \bar{b}_n \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{then}\ \bar{v} \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{else let} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{defs}_1 \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{in if}\ \hat{}s \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \texttt{then}\ \bar{v} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \texttt{else let} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{defs}_2 \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \texttt{in}\ \mathbf{TI}[\![R]\!][\![f]\!][\![e]\!] \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad [\![F(t_1,\ \ldots,\ t_m,\ x_1\texttt{+}\bar{c}_1,\ \ldots,\ x_n\texttt{+}\bar{c}_n,\ r_1,\ \ldots,\ r_o)]\!] \\
\quad \bar{b}_j\ \texttt{=}b_j\texttt{;}\ \bar{c}_j\ \texttt{=}\ c_j\texttt{;}\ \bar{v}\ \texttt{=}\ v \\
\texttt{in} \\
\quad F(l_1,\ \ldots,\ l_m,\ a_1,\ \ldots,\ a_n,\ u_1,\ \ldots,\ u_o)
\end{array}\right]\!\!\right]$$

Figure 4.4: Standard Composition

returns a copy of l with all elements removed for which p does not hold.

```
pairs(l1,l2) = list [ (x1,x2) | x1 <- l1 | x2 <- l2 ]
```

returns the “cross product” of the lists l_1 and l_2.

```
zip(l1,l2) = list [ (x1, x2) | x1 <- l1, x2 <- l2 ]
```

merges the lists l_1 and l_2 into one.

```
forall(p,l) = comp(and, True) [ p(x) | x <- l ]
```

computes $\forall x \in l.p(x)$.

```
append(l1,l2) = comp(cons, l2) [ x | x <- l1 ]
```

appends the lists l_1 and l_2.

```
flatten(l) = comp(append, [ ]) [ x | x <- l ]
```

flattens l by appending all sublists in l.

$\mathbf{TC}[\![$`rcomp`(f, b) `[` e `|` I `]`$]\!]$ = $\mathbf{TJ}[\![I]\!][\![f]\!][\![e]\!][\![b]\!]$

$\mathbf{TJ}[\![\ ]\!][\![f]\!][\![e]\!][\![v]\!] = [\![f(e, v)]\!]$

$\mathbf{TJ}[\![h_i$`:`t_i`<-`l_i`,` x_j`<-`a_j`..`b_j`+=`c_j`,` y_k`=`u_k`,` $defs_1$`;` s`;` y_k`' =` r_k`,` $defs_2$ `|` $R]\!][\![f]\!][\![e]\!][\![v]\!]$ =

```
let
  F(z1, ..., zm, x1, ..., xn, y1, ..., yo, w) =
    case z1 of
      [ ] => w
      h1:t1 =>
        ...
          case zm of
            [ ] => w
            hm:tm =>
              if x1 > b̄1 || ... || xn > b̄n
                then v̄
                else let
                  defs1
                in if ^s
                  then v̄
                  else let
                    defs2
                  in F(t1, ..., tm, x1+c̄1, ..., xn+c̄n, r1, ..., ro,
                      TJ⟦R⟧⟦f⟧⟦e⟧⟦v⟧)
  b̄j =bj; c̄j = cj
in
  F(l1, ..., lm, a1, ..., an, u1, ..., uo, v)
```

Figure 4.5: Reverse Composition

```
imult(x,y) = rcomp(iadd, izero) [ idmult(x,yi) | yi <- y ]
```

multiplies two integers x and y represented as lists of digits by multiplying each digit y_i with x and adding the products.

```
split(e,l) = rcomp(choose(e), (Nil, Nil)) [ x | x <- l ]
choose(e,x,(a,b)) = if x < e then (x:a,b) else (a,x:b)
```

splits the list l into two sublists of all elements less than e respectively all elements larger than e (in the reverse order).

```
split(e,l) = loop((Nil, Nil)) [ (a,b) |
  x <- l, a = [ ], b = [ ] ;;
  a' = if x > e then x:a else a,
  b' = if x > e then b else x:b ]
```

does exactly the same.

The example program in Appendix C.1 contains several more complicated applications of iterated compositions.

4.2.3 Translation

Instead of constructing the denotational semantics of iterated compositions, we will define a translation of these expressions into the core language. The basic idea is that each $iter_i$ is translated into a recursive function that is called with the actual bindings of the iterated variables. A sequence $iter_1$ | ... | $iter_n$ is translated into n nested recursive functions (that are lifted to the top-level of the program as shown in Subsection 4.3.3).

The translation function **TC** is based on **TI** that is defined by recursion over the number of iterators n. We assume that all sorts of iterators and definitions have been identified and grouped together. Figure 4.4 shows the translation of standard composition.

F is a new function name that is nowhere else used. Like for l_i, a_j and r_k, the values of $\bar{b}_j, \bar{v}_j$ and $\bar{v}$ are determined in the current environment. $defs_1$, s and $defs_2$ may refer to h_i, t_i, x_j and y_k and are evaluated in the denoted order. If an arithmetic iterator contains `-=` instead of `+=`, `+` is replaced by `-` and `>` by `<`.

The translation of reverse composition shown in Figure 4.5 differs from standard composition in that the new function F gets an additional parameter to denote the base case of recursion and in the construction of the (tail-)recursive function call.

As an example for the translation of an iterated composition take the `fiter` function defined in the previous subsection. Its translation yields (after some beautifying)

```
filter(p, l) =
  case l of
    [ ] => [ ]
    x:t =>
      if p(x)
        then x : filter(p, t)
        else filter(p, t)
```

The iterated composition consists of two iterator levels. However the inner one is just the test of the condition $p(x)$ but does not introduce any new iterated variable bindings. In this case the compiler does not introduce a new function definition but directly inserts the conditional expression into the outer function.

As an example for the translation of a doubly nested iterated composition take the function `pairs` in the previous subsection. Its translation is

```
pairs(l1, l2) =
  let
    pairs1(l1) =
      case l1 of
        [ ] => [ ]
        x1:t1 =>
          let
            pairs2(l2) =
              case l2 of
                [ ] => pairs1(t1)
                x2:t2 => (x1,x2) : pairs2(t2)
          in pairs2(l2)
  in pairs1(l1)
```

After lambda-lifting (see Subsection 4.3.3) and some inlining we get

```
pairs(l1, l2) =
  case l1 of
    [ ] => [ ]
    x1:t1 => pairs2(l2, l2, x1, t1)
pairs2(l, l2, x1, t1) =
  case l of
    [ ] => pairs(t1, l2)
    x2:t2 => (x1,x2) : pairs2(t2, l2, x1, t1)
```

4.3 Program Transformations

This section describes how we transform a program in the full language into a program in the core language. The full language provides limited higher order capabilities, has a polymorphic type system, and allows nested function definitions. The core language is strictly first-order, monomorphic, and contains combinator definitions only.

There are also several transformations whose purpose is to improve the efficiency of the program (see Subsection 4.3.5).

4.3.1 Uncurrying

A *curried* function application provides fewer arguments than the definition actually demands. The result of this application is a function that is specialized with respect to these arguments. E.g. the program

```
let
  map(f,l) = comp(f,[ ]) [ x | x <- l ]
  f(x,y,z) = x+y*z
in
  map(f(a,b), l)
```

applies the unary function $\lambda x.a + b * x$ to each element of l. The compiler transforms this program into

```
let
  map(f,l) = comp(f,[ ]) [ x | x <- l ]
  f(x,y,z) = x+y*z
  fxy(z) = f(a,b,z)
in
  map(fxy, l)
```

More formally, we apply the following transformation:

$$\mathbf{UC}\left[\!\!\left[\begin{array}{ll} \texttt{let} & \ldots \\ & f(x_1, \ldots, x_n) \;\texttt{=}\; exp \\ & \ldots \\ \texttt{in} & \boxed{\begin{array}{l} \ddots \\ \qquad f(a_1, \ldots, a_{m<n}) \end{array}} \end{array}\right]\!\!\right] =$$

$$\left[\!\!\left[\begin{array}{ll}\texttt{let} & \ldots \\ & f(x_1,\ldots,x_n) \;\texttt{=}\; exp \\ & \ldots \\ \texttt{in} & \\ & \boxed{\begin{array}{l}\ddots \\ \qquad \texttt{let}\;\; \bar{f}(x_{m+1},\ldots,x_n) \;\texttt{=}\; f(a_1\ldots a_m, x_{m+1}\ldots x_n) \\ \qquad \texttt{in}\;\;\; \bar{f}\end{array}}\end{array}\right]\!\!\right]$$

Since curried function names may only occur as arguments of higher order functions, the definition of the new function $\bar{f}$ may be lifted out of this application.

4.3.2 Specialization

The language allows to parameterize functions by other functions (see the example in the previous subsection). This form of a higher-order function is treated as a *function template* that is instantiated at compile-time e.g.

```
let
  map(f,l) = comp(f,[ ]) [ x | x <- l ]
  f(x,y,z) = x+y*z
  fxy(z) = f(a,b,z)
  mapfxy(l) = comp(fxy,[ ]) [ x | x <- l ]
in
  mapfxy(l)
```

The transformation process is described formally by

$$\mathbf{SP}\left[\!\!\left[\begin{array}{ll}\texttt{let} & \ldots \\ & f(x_1,\ldots,x_n) \;\texttt{=}\; exp \\ & \ldots \\ \texttt{in} & \\ & \boxed{\begin{array}{l}\ddots \\ \qquad f(a_1\ldots a_{i-1},\; g,\; a_{i+1}\ldots a_n)\end{array}}\end{array}\right]\!\!\right] =$$

$$\left[\!\!\left[\begin{array}{ll}\texttt{let} & \ldots \\ & f(x_1,\ldots,x_n) \;\texttt{=}\; exp \\ & \ldots \\ \texttt{in} & \\ & \boxed{\begin{array}{l}\ddots \\ \qquad \texttt{let}\;\; \bar{f}(x_1\ldots x_{i-1}, x_{i+1}\ldots x_n) \;\texttt{=}\; \mathbf{TS}[\![exp]\!][[\![x_i]\!] \leftarrow [\![g]\!]] \\ \qquad \texttt{in}\;\;\; \bar{f}(a_1\ldots a_{i-1}, a_{i+1}\ldots a_n)\end{array}}\end{array}\right]\!\!\right]$$

TS just replaces all occurrences of x_i by g. After the transformation of all applications, the original higher order function is removed from the program.

4.3.3 Lambda Lifting

The body of a function definition may contain references to constant names that are defined in the global environment e.g.

```
let
  f(x) = x+a
  a = 1
in f(7)
```

For the compilation it is convenient to have only *combinators* i.e. functions whose bodies are "closed" with respect to their parameters. The result of a combinator only depends on its arguments, thus combinators can be easily manipulated and combined (and evaluated on a remote processor).

In order to transform a function into a combinator, we have to add all global variables referenced in the function body to the parameters of the function and to the arguments of each application i.e.

```
let f(x,a) = x+a in
let a = 1
in f(7,a)
```

The situation becomes more complicated if a function references another function defined in the same block e.g.

```
let
  f(x) = a+g(x)
  g(y) = if y < 0 then b else f(y-1)
  a = 1
  b = 2
in
  f(7)
```

In general we have to add as arguments all global variables that are referenced by any function defined in the same block and (directly or indirectly) called by the function under consideration:

```
let
  f(x,a,b) = a+g(x,b)
  g(y,a,b) = if y < 0 then b else f(y-1,a,b)
in let
  a = 1
  b = 2
in
  f(7,a,b)
```

More formally, we define the lambda lifting process by a sequence of transformations to be applied to expressions inside out.

Separation

First we separate function definitions from constant definitions lifting the function definitions to the outer level.

$$\mathbf{LL}\left[\!\!\left[\begin{array}{ll} \texttt{let} & \ldots \\ & f_i(p_{i1}\ldots p_{in_i}) = def_i \\ & \ldots \\ & c_j = exp_j \\ & \ldots \\ \texttt{in} & exp \end{array}\right]\!\!\right] =$$

$$\left[\!\!\left[\begin{array}{lll}\texttt{let} & \ldots & \\ & f_i\,(p_{i1}\ldots p_{in_i}, q_{i1}\ldots q_{im_i}) \texttt{ = } \mathbf{TL}[\![\mathit{def}_i]\!][\ldots f_i \leftarrow q_i \ldots] & \\ & \ldots & \\ \texttt{in} & \texttt{let} \;\ldots & \\ & \quad c_j \texttt{ = } \mathbf{TL}[\![\mathit{exp}_j]\!][\ldots f_i \leftarrow q_i \ldots] & \\ & \quad \ldots & \\ & \texttt{in} \;\; \mathbf{TL}[\![\mathit{exp}]\!][\ldots f_i \leftarrow q_i \ldots] & \end{array}\right]\!\!\right]$$

where

$$q = \mathrm{seq}\{[\![c_j]\!] | [\![c_j]\!] \in \mathbf{FV}[\![\mathit{def}_i]\!] \cup \{\mathbf{FV}[\![\mathit{def}_k]\!] | [\![\mathit{def}_i]\!] \rightarrow^*_{FF} [\![\mathit{def}_k]\!]\}\}$$
$$[\![\mathit{def}_i]\!] \rightarrow_{FF} [\![\mathit{def}_j]\!] \Leftrightarrow f_j \in \mathbf{FF}[\![\mathit{def}_i]\!]$$

i.e. q is the sequence of free variables to become function arguments, $\rightarrow_{FF}$ is the relation "contains a call of the function with body" and $\rightarrow^*_{FF}$ is the transitive closure of this relation. $\mathbf{TL}[\![\mathit{exp}]\!][\ldots f_i \leftarrow q \ldots]$ appends in $[\![\mathit{exp}]\!]$ the argument list q to each application of f_i. The transformed f_i do not any more reference the c_j defined in the block and may be lifted to the outer level.

Constant Context

Second if a block of lifted function definitions is enclosed in an outer block of constant definitions, we interchange the nesting of blocks by applying the transformation

$$\mathbf{LL}\left[\!\!\left[\begin{array}{ll}\texttt{let} & \ldots \\ & c_j \texttt{ = } \ldots \\ & \ldots \\ \texttt{in} & \\ & \boxed{\begin{array}{l}\ddots \\ \qquad \texttt{let} \;\ldots \\ \qquad\qquad f_i\,(p_{i1}\ldots p_{in_i}) \texttt{ = } \mathit{def}_i \\ \qquad\qquad \ldots \\ \qquad \texttt{in}\; \mathit{exp}\end{array}}\end{array}\right]\!\!\right] =$$

$$\left[\!\!\left[\begin{array}{ll}\texttt{let} & \ldots \\ & f'_i\,(p_{i1}\ldots p_{in_i}, q_{i1}\ldots q_{im_i}) \texttt{ = } \mathbf{TL}[\![\mathit{def}_i]\!][\ldots f_i \leftarrow (f'_i, q_i) \ldots] \\ & \ldots \\ \texttt{in} & \texttt{let} \;\ldots \\ & \quad c_j \texttt{ = } \ldots \\ & \quad \ldots \\ & \texttt{in} \\ & \quad \boxed{\begin{array}{l}\ddots \\ \qquad \mathbf{TL}[\![\mathit{exp}]\!][\ldots f_i \leftarrow (f'_i, q_i) \ldots]\end{array}}\end{array}\right]\!\!\right]$$

In order to resolve name clashes, the lifted version of a function f_i has to be renamed into f'_i. Thus **TL** has to rename function applications as well.

Function Context

Third if a block of function definitions is contained in another function definition, we draw the inner definitions to the level of the outer one (the global "constants" c_i are the parameters of the outer function):

$$\mathbf{LL}\left[\!\!\left[\begin{array}{l} \texttt{let}\;\ldots \\ \quad f(\ldots c_j \ldots)\; \texttt{=} \\ \quad \boxed{\begin{array}{l} \ddots \\ \quad \texttt{let}\;\ldots \\ \qquad f_i(p_{i1}\ldots p_{in_i})\; \texttt{=}\; \mathit{def}_i \\ \qquad \ldots \\ \quad \texttt{in}\; \mathit{exp} \end{array}} \\ \quad \ldots \\ \texttt{in}\;\ldots \end{array}\right]\!\!\right] \;\texttt{=}$$

$$\left[\!\!\left[\begin{array}{l} \texttt{let}\;\ldots \\ \quad f(c_1\ldots c_m)\; \texttt{=} \\ \quad \boxed{\begin{array}{l} \ddots \\ \quad \mathbf{TL}[\![\mathit{exp}]\!][\ldots f_i \leftarrow (f'_i, q_i)\ldots] \end{array}} \\ \quad \ldots \\ \quad f'_i(p_{i1}\ldots p_{in_i}, q_{i1}\ldots q_{im_i})\; \texttt{=}\; \mathbf{TL}[\![\mathit{def}_i]\!][\ldots f_i \leftarrow (f'_i, q_i)\ldots] \\ \quad \ldots \\ \texttt{in}\;\ldots \end{array}\right]\!\!\right]$$

If none of the two previous transformations can be applied any more, all functions have been transformed into combinators that can be collected in a block at the top-level of the program (as demanded in the abstract syntax of the core language).

4.3.4 Monomorphization

The language allows the definition of polymorphic functions e.g.

```
let
  prod :: ([*a], [*b]) -> [(*a,*b)]
  prod(l1,l2) = list [ (x,y) | x <- l1 | y <- l2 ]
in
  prod([1,2,3], [True, False])
```

The compiler transforms each application of a polymorphic function into the application of a monomorphic instance i.e.

```
let
  prod :: ([*a], [*b]) -> [(*a,*b)]
  prod(l1,l2) = list [ (x,y) | x <- l1 | y <- l2 ]
  ibprod :: ([Int], [Bool]) -> [(Int,Bool)]
  ibprod(l1,l2) = list [ (x,y) | x <- l1 | y <- l2 ]
in
  ibprod([1,2,3], [True, False])
```

The type of the function instance is derived from the types of the given arguments. This is possible by a simple matching algorithm provided that all type variables of the image type (`[(*a,*b)]` in the example above) are among the type variables of the domain type (`([*a], [*b])`).

The only problem arises if the polymorpic constants `[ ]` or `Error` are applied to a polymorphic function definition. E.g. the application

```
prod([ ], [True, False])
```

cannot be typed by the compiler. A solution in such a (rather rare) case is to define a named constant with a monomorphic type signature:

```
let
  inil :: [Int]
  inil = [ ]
in
  prod(inil, [True, False])
```

After the transformation of all polymorphic function applications the polymorphic function definition is removed from the program.

4.3.5 Optimizing Transformations

The compiler performs several program transformations whose purpose is to improve the efficiency of the program.

Function Inlining

The compiler inlines the body of every not (directly or indirectly) recursive function into each application of this function:

$$\mathbf{IL}\left[\!\!\left[\begin{array}{ll}\texttt{let} & \ldots \\ & f(p_1 \ldots p_n)\ \texttt{=}\ \mathit{def} \\ & \ldots \\ \texttt{in} & \boxed{\begin{array}{l}\ddots \\ \qquad f(a_1 \ldots a_n)\end{array}}\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{ll}\texttt{let} & \ldots \\ & \ldots \\ \texttt{in} & \\ & \boxed{\begin{array}{lll}\ddots & & \\ & \texttt{let} & p_1\ \texttt{=}\ a_1 \\ & & \ldots \\ & & p_n\ \texttt{=}\ a_n \\ & \texttt{in} & \mathit{def}\end{array}}\end{array}\right]\!\!\right]$$

Like the function application, the `let` construct is strict in the values a_i, thus the termination properties of the program remain unchanged. After inlining, the function definition is removed from the program.

In the body of a mutually recursive function, the compiler also inlines the call of another function if this function is a member of the current "recursion cycle" but is itself not directly recursive. Consequently, the recursion cycle may collapse into a single recursive function (e.g. if it is of the form $f_1 \rightarrow f_2 \rightarrow \ldots \rightarrow f_n \rightarrow f_1$ and each $f_{i \neq 1}$ is neither directly recursive nor member of another cycle).

Function inlining increases the size of the program but reduces the number of function calls and allows several further optimizations to be applied.

Constant Inlining

The compiler inlines the value of a constant definition into a reference to this constant, if it is the only one:

$$\mathbf{IL}\left[\!\!\left[\begin{array}{ll}\texttt{let} & \ldots \\ & c\ \texttt{=}\ exp \\ & \ldots \\ \texttt{in} & \\ & \boxed{\begin{array}{r}\ddots\ \ \ \\ c\end{array}}\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{ll}\texttt{let} & \ldots \\ & \ldots \\ \texttt{in} & \\ & exp\end{array}\right]\!\!\right]$$

Since the constant is only referenced once, there is no danger that the value of the constant is computed more than once. The constant definition is finally removed from the program.

Please note that by this optimization we actually slightly change the denotation of the program. Consider e.g. the transformation

$$\mathbf{IL}\left[\!\!\left[\begin{array}{l}\texttt{let a = f(b)} \\ \texttt{in if\ \ b > 0} \\ \texttt{\ \ \ \ \ \ \ then 1} \\ \texttt{\ \ \ \ \ \ \ else g(a)}\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\texttt{if\ \ b > 0} \\ \texttt{\ \ \ \ then 1} \\ \texttt{\ \ \ \ else g(f(b))}\end{array}\right]\!\!\right]$$

If the evaluation of `f(b)` runs into non-termination/error, then the original program runs into non-termination/error, too. However, the transformed program returns 1 for $b > 0$. Actually this would be also the result of the original program when evaluated under a non-strict (e.g. lazy) evaluation strategy. Thus we have

$$\mathbf{P}[\![program]\!] \sqsubseteq \mathbf{C}[\![program]\!] \sqsubseteq \mathbf{Q}[\![program]\!]$$

where **P** is the denotation defined in Section 4.1, **C** is the semantic actually implemented by the compiler and **Q** is a fictive non-strict interpretation of the language. We therefore should not inline a constant that is referenced in a non-strict context (like a conditional). However, currently the compiler is sloppy and omits the strictness check.

Expression Folding

After the inlining phase, the compiler replaces several forms of compound expressions by equivalent but simpler ones. Expressions amenable to folding are for instance

1. *Arithmetic Expressions*

   $$\mathbf{CF}[\![int_1\texttt{*}int_2]\!] = [\![\mathbf{I}[\![int_1]\!] * \mathbf{I}[\![int_2]\!]]\!]$$

2. *Logical Expressions*

   $$\mathbf{CF}[\![exp\ \texttt{\&\& True}]\!] = [\![exp]\!]$$

3. *Conditionals*

   $$\mathbf{CF}[\![\texttt{if False then}\ exp_1\ \texttt{else}\ exp_2]\!] = [\![exp_2]\!]$$

4. *List Selections*

   $$\mathbf{CF}[\![\texttt{case}\ exp_h\texttt{:}exp_t\ \texttt{of [ ] =>}\ exp_n\texttt{;}\ id_h\texttt{:}id_t\ \texttt{=>}\ exp_c]\!] =$$
   $$[\![\texttt{let}\ id_h\ \texttt{=}\ exp_h\texttt{;}\ id_t\ \texttt{=}\ exp_t\ \texttt{in}\ exp_c]\!]$$

5. *Tuple Selections*

$$\mathbf{CF}[\![\texttt{(}exp_1\texttt{,}\ \ldots\ \texttt{,}\ exp_n\texttt{)}\texttt{.}int]\!] = [\![exp_{\mathbf{I}[\![int]\!]}]\!]$$

Several of these transformations change the semantics of the program as described in the previous paragraph. For instance, the transformation

$$\mathbf{CF}[\![\texttt{0}\ \texttt{*}\ exp]\!] = [\![0]\!]$$

is actually only valid under the assumption that the evaluation of *exp* returns a result. Again, we value the practical optimization effect more than the theoretical change in the program semantics.

Condition Drawing

In the code generation phase, several optimizations are applied on sequences of assignment statements. These optimizations may dramatically affect the performance of a program by transforming e.g. a recursive function into a loop. If a conditional interrupts such a sequence, these optimizations may be prohibited.

For this reason we attempt to draw conditional expressions to the outermost level thus increasing the size of unconditionally evaluated expressions:

$$\mathbf{CD}\left[\!\!\left[\ \boxed{\begin{array}{lll} \ddots & & \\ & \texttt{if} & exp_c \\ & & \texttt{then}\ exp_t \\ & & \texttt{else}\ exp_f \end{array}}\ \right]\!\!\right] = \left[\!\!\left[\begin{array}{ll} \texttt{if} & exp_c \\ & \texttt{then} \\ & \boxed{\begin{array}{ll} \ddots & \\ & exp_t \end{array}} \\ & \texttt{else} \\ & \boxed{\begin{array}{ll} \ddots & \\ & exp_f \end{array}} \end{array}\right]\!\!\right]$$

We only apply this transformation if the context expression does not contain another conditional expression whose test condition encloses the conditional to be lifted. Otherwise we would interchange the nesting of conditionals and the transformation would be applied in a loop forever.

Furthermore, the transformation is only valid if the context expression does not contain a block of local definitions whose body expression encloses the conditional. Otherwise such a block might define variables that are referenced in the condition exp_c.

If a conditional is enclosed by a local definition block, we can still lift the conditional somewhat further assuming that some of the defined variables are not referenced in exp_c:

$$\mathbf{CD}\left[\!\!\left[\begin{array}{lll} \texttt{let} & \ldots & \\ & c_i\ \texttt{=}\ def_i & \\ & \ldots & \\ \texttt{in} & \texttt{if} & exp_c \\ & & \texttt{then}\ exp_t \\ & & \texttt{else}\ exp_f \end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{lllll} \texttt{let} & \ldots & & & \\ & c_i\ \texttt{=}\ def_i & & & \\ & \ldots & & & \\ \texttt{in} & \texttt{if} & exp_c & & \\ & & \texttt{then} & \texttt{let} & \ldots \\ & & & & c_j\ \texttt{=}\ def_j \\ & & & & \ldots \\ & & & \texttt{in} & exp_t \\ & & \texttt{else} & \texttt{let} & \ldots \\ & & & & c_j\ \texttt{=}\ def_j \\ & & & & \ldots \\ & & & \texttt{in} & exp_f \end{array}\right]\!\!\right]$$

The c_j are those variables that are not referenced in exp_c and that are not used in the definition of any variable referenced in exp_c. Drawing the conditional beyond this block is not possible any more.

4.3.6 Final Processing

As the last step before code generation we finally transform the program representation from an expression into a set of function definitions:

$$\mathbf{FP}\left[\begin{array}{l}\texttt{let}\ \mathit{fdefs}\\ \texttt{in}\ \mathit{exp}\end{array}\right] = \left[\begin{array}{l}\mathit{fdefs}\\ \texttt{dcmain() = }\mathit{exp}\end{array}\right]$$

The parameter-less main function `dcmain` will become the external entry point to the program; for more details on the external interface see Section 4.5.

4.4 Code Generation and Optimization

In this section we describe the techniques that we apply for generating efficient C code from programs in the functional core language.

4.4.1 Basic Code Generation

The compiler has a small and well defined interface to the runtime system that consists of a number of basic types and of the constructor and selector functions for lists and for tuples. The compiler does not care for memory management and garbage collection at all but leaves this task entirely to the runtime system (see Section 4.5).

After the transformation phase, the program consists of a set of closed function definitions. Each function but the main function is (directly or indirectly) recursive and each local constant is referenced at least twice. Otherwise, the function respectively constant would have been inlined and the definition would have been removed.

We translate each function in the core language into a C function. The main difficulty of this translation arises from the fact that the body of a C function consists of a *statement sequence* while the body of a function in the core language is an *expression*. The compilation of a function body is therefore a mapping from expressions to statement sequences.

This mapping is trivial if we are able to map a functional expression *exp* into a corresponding C expression *cexp* and define the body of the C function just as the statement `return(`*cexp*`)`. However, there are several functional expressions that cannot be translated into C expressions but have to be compiled into statement sequences preceding the `return` statement:

- *Local Definitions:* A block of local definitions is compiled into a sequence of variable declarations and assignments. This sequentialization is possible since constant definitions have been topologically sorted in the transformation phase. To make the generated C code more compact, variable definitions and assignment statements are combined to initialized variable declarations.

  Depending on the context of the definition block, the last statement in the generated sequence is either a `return` (if the result of the block is also the result of the function) or the assignment of the result to a variable (if the result is the value of a named constant). If the result is used anonymously, an auxiliary identifier is introduced to receive the result.

The following rules show the translation of a definition block that represents the body of a function (where t_f represents the result type of f and t_x represents the type of a constant respectively parameter x)

$$
\mathbf{DC}\left[\!\!\left[\begin{array}{l}
f\ (\ldots p_i \ldots)\ \texttt{=}\\
\quad\texttt{let}\\
\qquad\ldots\\
\qquad c_j\ \texttt{=}\ exp_j\\
\qquad\ldots\\
\quad\texttt{in}\\
\qquad exp
\end{array}\right]\!\!\right]
\texttt{=}
\left[\!\!\left[\begin{array}{l}
t_f\ f\ (\ldots t_{p_i}\ p_i \ldots)\ \texttt{\{}\\
\quad\ldots\\
\quad t_{c_j}\ c_j\ \texttt{=}\ exp_j\texttt{;}\\
\quad\ldots\\
\quad\texttt{return(}exp\texttt{);}\\
\texttt{\}}
\end{array}\right]\!\!\right]
$$

and the translation of a function block whose result is used anonymously.

$$
\mathbf{DC}\left[\!\!\left[\boxed{\begin{array}{l}
\ddots\\
\qquad\texttt{let}\\
\qquad\quad\ldots\\
\qquad\quad {c_j}^{t_{c_j}} = exp_j\\
\qquad\quad\ldots\\
\qquad\texttt{in}\\
\qquad\quad exp
\end{array}}\right]\!\!\right]
\texttt{=}
\left[\!\!\left[\begin{array}{l}
\ldots\\
t_{c_j}\ c_j\ \texttt{=}\ exp_j\texttt{;}\\
\ldots\\
t_{exp}\ x\ \texttt{=}\ exp\texttt{;}\\
\boxed{\begin{array}{l}\ddots\\ \qquad\quad x\end{array}}
\end{array}\right]\!\!\right]
$$

During the code generation the context of an expression is propagated to the base expression of a block i.e. if `let` ... `in` *exp* is compiled in the context "function body", then all statement sequences generated from *exp* will be terminated by a **`return`**.

- *Conditional Expressions:* A conditional expression is compiled to the `if` statement of the C language. C also offers conditional expressions of the syntactic form c `?` t `:` f. However, we do not want to rely on this feature since it is not available in many other imperative languages and actually not typical for C programs written by humans.

  Each branch of the conditional expression is translated into a statement sequence propagating the context of the conditional expression: if the conditional expression is the body of the function, both generated statement sequences are terminated by a `return`. Otherwise, both branches finally assign their result to a (possibly auxiliary) variable.

  We show the translation of a conditional function body

$$
\mathbf{DC}\left[\!\!\left[\begin{array}{l}
f\ (\ldots p_i \ldots)\ \texttt{=}\\
\quad\texttt{if}\ c\\
\qquad\texttt{then}\ \ldots\\
\qquad\texttt{else}\ \ldots
\end{array}\right]\!\!\right]
\texttt{=}
\left[\!\!\left[\begin{array}{l}
t_f\ f\ (\ldots t_{p_i}\ p_i \ldots)\ \texttt{\{}\\
\quad\texttt{if (}c\texttt{) \{}\\
\qquad\ldots\\
\qquad\texttt{return(}\ldots\texttt{);}\\
\quad\texttt{\}}\\
\quad\texttt{else \{}\\
\qquad\ldots\\
\qquad\texttt{return(}\ldots\texttt{);}\\
\quad\texttt{\}}\\
\texttt{\}}
\end{array}\right]\!\!\right]
$$

and of a conditional expression that is used anonymously:

$$
\mathbf{DC}\left[\!\!\left[\ \boxed{\begin{array}{l}\ddots\\ \quad\ \texttt{if}\ c\\ \quad\quad\ \texttt{then}\ \ldots\\ \quad\quad\ \texttt{else}\ \ldots\end{array}}\ \right]\!\!\right] \;=\; \left[\!\!\left[\ \begin{array}{l} t\ x\texttt{;}\\ \ldots\\ \texttt{if (}c\texttt{) \{}\\ \quad\ldots\\ \quad x \texttt{ = } \ldots\texttt{ ;}\\ \texttt{\}}\\ \texttt{else \{}\\ \quad\ldots\\ \quad x \texttt{ = } \ldots\texttt{ ;}\\ \texttt{\}}\\ \boxed{\begin{array}{l}\ddots\\ \quad\quad x\end{array}} \end{array}\ \right]\!\!\right]
$$

- *Error Expressions:* Any strict expression containing the constant `Error`(*string*) is translated into a single statement `ERROR`(*string*) that aborts the execution of a program. To make sense such expressions must thus be encapsulated by conditional expressions:

$$
\mathbf{DC}\left[\!\!\left[\ \begin{array}{l}\texttt{if}\ c\\ \quad\texttt{then}\ \ldots\\ \quad\texttt{else}\ \ldots\texttt{Error(}string\texttt{)}\ldots\end{array}\ \right]\!\!\right] \;=\; \left[\!\!\left[\ \begin{array}{l} t\ x\texttt{;}\\ \texttt{if (}c\texttt{) \{}\\ \quad\ldots\\ \quad x \texttt{ = } \ldots\texttt{ ;}\\ \texttt{\}}\\ \texttt{else}\\ \quad\texttt{ERROR(}string\texttt{)} \end{array}\ \right]\!\!\right]
$$

The result of the basic code generation phase is a program in the C language consisting of a set of recursive function definitions. The remaining sections discuss how to improve the time and memory efficiency of the program by transforming recursion into iteration.

4.4.2 Tail-Recursion

Recursion is a rather expensive form of expressing repetitive computations since a recursive function consumes $O(n)$ memory where n is the depth of the call tree. A key strategy to improve the efficiency of a functional program is therefore to transform a recursive function into an iterative loop that copes with $O(1)$ memory and is also considerably faster. In particular, there exists the class of *tail-recursive* functions which can be easily transformed into loops in an automatic fashion.

A tail-recursive function call is a recursive function call whose result is directly forwarded to the caller of the function. E.g. in the function

```
f(x, y) =
  if x == 0
    then y
    else if even(x)
           then 1+f(x-1,y-1)
           else f(x-1,y+1)
```

the call `f(x-1,y+1)` is tail-recursive. A function is called tail-recursive, if all of its recursive calls are tail-recursive.

Above function is compiled into the following C function:

```
int f(int x, int y) {
  if (x == 0)
    return(y);
  else if (even(x))
    return(1+f(x-1,y-1));
  else
    return(f(x-1,y+1));
}
```

The tail-recursive call in this function can be translated into the iteration of a while loop

```
int f(int x, int y) {
  while (TRUE) {
    if (x == 0)
      return(y);
    else if (even(x))
      return(1+f(x-1,y-1));
    else {
      x = x-1;
      y = y+1;
    }
  }
}
```

by destructively updating of function parameters by the arguments of the call. The current function activation performs the job of the tail-recursive one by a straight jump to the begin of the function body.

A tail-recursive function can therefore be transformed into a function that is not recursive at all. However, even if there remain some recursive function calls that are not tail-recursive (as in above example), the depth of the call tree may be considerably decreased.

The jump to the begin of the function can be implemented by a `goto` statement but we prefer the more structured form of a `while` loop. After some beautifying we get the form

```
int f(int x, int y) {
  while (x != 0) {
    if (even(x))
      return(1+f(x-1,y-1));
    else {
      x = x-1;
      y = y+1;
    }
  }
  return(y);
}
```

that a human programmers would actually write.

The transformation is therefore fairly easy, the only problem arises in determining the order in which the parameters are to be updated for the next iteration of a loop. A tail-recursive function call

```
f(x, y) {
  ...
  f(y, x-1)
  ...
}
```

must be compiled into

```
f(x, y) {
  while (TRUE) {
    ...
    z = x
    x = y
    y = z-1
    ...
  }
}
```

where `z` is an auxiliary variable that has to be introduced since there exists a dependency from `x` to `y` and vice versa. More formally, we apply the transformation

$$\mathbf{TR}\left[\!\!\left[\begin{array}{l} t_f\ f(t_1\ x_1,\ \ldots\ ,\ t_n\ x_n)\ \texttt{\{} \\ \quad \ldots \\ \quad \boxed{\texttt{return}(f(a_1,\ \ldots\ ,\ a_n));} \\ \quad \ldots \\ \texttt{\}} \end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l} t_f\ f(t_1\ x_1,\ \ldots\ ,\ t_n\ x_n)\ \texttt{\{} \\ \quad \texttt{while (TRUE) \{} \\ \qquad \ldots \\ \qquad \boxed{\begin{array}{l} x_{p_1}\ \texttt{=}\ a_{p_1}; \\ \ldots \\ x_{p_n}\ \texttt{=}\ a_{p_n}; \end{array}} \\ \qquad \ldots \\ \quad \texttt{\}} \\ \texttt{\}} \end{array}\right]\!\!\right]$$

where p is a topological order of the graph

$$(V, E) = (\{x_1, \ldots, x_n\}, \{(x_i, x_j) | i \neq j \wedge x_i \rightarrow_R x_j\}).$$

$x_i \rightarrow_R x_j$ stands for "x_j is used in the new value for x_i". If such an order does not exist, some dependency must be removed by adding a new variable z to V and replacing each dependence $x_i \rightarrow x_j$ in E by $x_i \rightarrow z$. In other words, an assignment z = x_j is added and all further assignments use z instead of x_j.

After the introduction of a loop we perform a "beautifying transformation"

$$\mathbf{BT}\left[\!\!\left[\begin{array}{l} t_f\ f(t_1\ x_1,\ \ldots\ ,\ t_n\ x_n)\ \texttt{\{} \\ \quad \texttt{while (TRUE) \{} \\ \qquad \texttt{if (}\mathit{cond}\texttt{) \{} \\ \qquad\quad \boxed{\begin{array}{l} \ldots \\ \texttt{return(}\mathit{exp}\texttt{);} \end{array}} \\ \qquad \texttt{\}} \\ \qquad \texttt{else \{} \\ \qquad\quad \ldots \\ \qquad \texttt{\}} \\ \quad \texttt{\}} \\ \texttt{\}} \end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l} t_f\ f(t_1\ x_1,\ \ldots\ ,\ t_n\ x_n)\ \texttt{\{} \\ \quad \texttt{while (!}\mathit{cond}\texttt{) \{} \\ \qquad \ldots \\ \quad \texttt{\}} \\ \quad \boxed{\begin{array}{l} \ldots \\ \texttt{return(}\mathit{exp}\texttt{);} \end{array}} \\ \texttt{\}} \end{array}\right]\!\!\right]$$

if possible.

The transformation of tail-recursive functions into loops is a standard technique in compilation of functional languages. However, many interesting functions are not tail-recursive. In the following subsections we describe two further optimization techniques that transform frequently occurring forms of recursion into tail-recursion. We are not aware of other compilers applying these forms of optimizations.

4.4.3 List Results

Consider the core-language function

```
f(x) =
  if x == 0
    then [ ]
    else g(x):f(x-1)
```

which is the general pattern for a large class of list-generating functions; e.g. it is generated from the iterated composition

```
list [ g(x) | x <- n..1 -= 1 ]
```

This function is compiled into the C function

```
intlist f(int x) {
  if (x == 0)
    return(NIL);
  else
    return(CONS(g(x), f(x-1)));
}
```

Unfortunately, the function is not tail-recursive and therefore not amenable to the transformation described in the previous section.

Of course the result list of above function might be constructed also tail-recursively in the reverse order and the tail-recursive function `reverse` might be applied to create the actual result. However this solution is inelegant and requires twice as much time and memory[2].

An experienced programmer will instead write the following function

```
intlist f(int x) {
  intlist y;
  intlist *z = &y;
  while (x != 0) {
    *z = NEW();
    (*z)->head = g(x);
    z = &(*z)->tail;
    x = x-1
  }
  *z = NIL;
  return(y);
}
```

[2]Actually several functions of the SACLIB system are implemented in this style.

that has the same input/output behaviour but copes with a simple while loop. The main idea is to remember in a variable `z` the address of the tail of the previous list cell which is updated with the pointer to next cell. The function `NEW` allocates such a cell.

In order to automatize this tranformation we introduce the operation

```
cell *RCONS(word x, cell *z) {
  *z = NEW();
  (*z)->head = x;
  return(&(*z)->tail);
}
```

Compare this function with the code above and also with the definition of the `CONS` operation for the construction of a new list cell:

```
cell CONS(word x, cell y) {
  cell z = NEW();
  z->head = x;
  z->tail = y;
  return(z);
}
```

The compiler now applies `RCONS` to transform the function `f` at the begin of this subsection into the tail-recursive form

```
void _f(int x, intlist *y) {
  if (x == 0)
    *y = NIL;
  else
    _f(x-1, RCONS(g(x), y));
}

intlist f(int x) {
  intlist y;
  _f(x, &y);
  return(y);
}
```

introducing a procedure `_f` with a tail-recursive call. After transforming this call into a loop we get

```
void _f(int x, intlist *y) {
  while (x != 0) {
    y = RCONS(g(x), y);
    x = x-1;
  }
  *y = NIL;
}

intlist f(int x) {
  intlist y;
  _f(x, &y);
  return(y);
}
```

which (after inlining `_f` and `RCONS`) is exactly the efficient program shown above.

More formally, the compiler applies the following transformation

$$
\mathbf{LR}\left[\!\!\left[\begin{array}{l}
t_f\ f(t_1\ x_1,\ \ldots\ ,\ t_n\ x_n)\ \texttt{\{} \\
\quad \ldots \\
\quad \boxed{\texttt{return}(b)\texttt{;}} \\
\quad \ldots \\
\quad \boxed{\texttt{return(CONS}(e,\ f(a_1,\ \ldots\ ,\ a_n)))\texttt{;}} \\
\quad \ldots \\
\texttt{\}}
\end{array}\right]\!\!\right] =
$$

$$
\left[\!\!\left[\begin{array}{l}
\texttt{void}\ g(t_1\ x1,\ \ldots\ ,\ t_n\ x_n,\ t_f\ *y)\ \texttt{\{} \\
\quad \ldots \\
\quad \boxed{\begin{array}{l} *y\ \texttt{=}\ b\texttt{;} \\ \texttt{return;} \end{array}} \\
\quad \ldots \\
\quad \boxed{\begin{array}{l} g(a_1,\ \ldots\ ,\ a_n,\ \texttt{RCONS}(e,\ y))\texttt{;} \\ \texttt{return;} \end{array}} \\
\quad \ldots \\
\texttt{\}} \\
\\
t_f\ f(t_1\ x_1,\ \ldots\ ,\ t_n\ x_n)\ \texttt{\{} \\
\quad t_f\ y\texttt{;} \\
\quad g(x_1,\ \ldots\ ,\ x_n,\ \texttt{\&}y)\texttt{;} \\
\quad \texttt{return}(y)\texttt{;} \\
\texttt{\}}
\end{array}\right]\!\!\right]
$$

The compiler applies the transformation whenever all recursive calls of a list-valued function are contained in a `return` call of above form.

4.4.4 Tuple Results

Take a look at the core language function

```
split(e, l) =
  case l of
    Nil => (Nil, Nil)
    h:t => let (a,b) = split(e, t)
           in if h < e
             then (h:a, b)
             else (a, h:b)
```

which splits a list into two result lists. The target code corresponding to this function is

```
listpair split(elem e, list l) {
  if (ISNIL(l))
    return(TUPLE2(NIL, NIL));
  else {
    elem h = HEAD(l);
    if (h < e) {
```

```
        listpair p = split(e, t);
        return(TUPLE2(CONS(h, ELEM1(p)), ELEM2(p)));
      }
      else {
        listpair p = split(e, t);
        return(TUPLE2(ELEM1(p), CONS(h, ELEM2(p))));
      }
    }
  }
```

Unfortunately the transformation of the previous subsection is not directly applicable to this function, since its result is a tuple object (however containing two lists). Instead the compiler transforms this function into

```
void split2(elem e, list l, list *l1, list *l2) {
  if (ISNIL(l)) {
    *l1 = NIL; *l2 = NIL;
  }
  else {
    elem h = HEAD(l);
    if (h < e) {
      list la, lb;
      split(e, TAIL(l), &la, &lb);
      *l1 = CONS(h, la); *l2 = lb;
    }
    else {
      list la, lb;
      split(e, TAIL(l), &la, &lb);
      *l1 = la; *l2 = CONS(h, lb);
    }
  }
}

listpair split(elem e, list l) {
  list l1, l2;
  split2(e, l, &l1, &l2);
  return(TUPLE2(l1, l2));
}
```

creating an auxiliary function that takes a reference argument for each field of the result tuple of the original function. Now a peephole optimizer generates

```
void split2(elem e, list l, list *l1, list *l2) {
  if (ISNIL(l)) {
    *l1 = NIL; *l2 = NIL;
  }
  else {
    elem h = HEAD(l);
    if (h < e)
      split(e, TAIL(l), RCONS(h, l1), l2);
    else
      split(e, TAIL(l), l1, RCONS(h, l2));
  }
```

```
}
```

which is amenable to tail-recursion removal. The final target code thus is

```
void split2(elem e, list l, list *l1, list *l2) {
  while (!ISNIL(l)) {
    elem h = HEAD(l);
    if (h < e) {
      l = TAIL(l);
      l1 = RCONS(h, l1);
    }
    else {
      l = TAIL(l);
      l2 = RCONS(h, l2);
    }
  }
  *l1 = NIL; *l2 = NIL;
}
```

which is as efficient as can be reasonably expected.

More formally, we apply the following transformation on a function f generating a tuple with component types $(s_1, \ldots, s_m)$:

```
TR [[ t_f f(t_1 x_1, ... , t_n x_n) {
        ...
        ... f(a_1, ... , a_n) ...
        ...
        return(b);
        ...
      }                                  ]] =

   [[ void g(t_1 x_1, ... , t_n x_n, s_1 *y_1, ... , s_m *y_m) {
        ...
        s_1 z_1; ... ; s_m z_m;
        ...
        g(a_1, ... , a_n, &z_1, ... , &z_m);
        ...
        ... TUPLE_m(z_1, ... , z_m) ...
        ...
          *y_1 = b.1;
          ...
          *y_m = b.m;
          return;
        ...
      }

      t_f f(t_1 x_1, ... , t_n x_n) {
        s_1 y_1; ... ; s_m y_m;
        g(x_1, ... , x_n, &y_1, ... , &y_m);
        return(TUPLE_m(y_1, ... , y_m));
      }                                  ]]
```

After this transformation, we apply the peephole optimizers

$$\mathbf{TR}\left[\!\!\left[\begin{array}{l}\texttt{void}\ f\texttt{(}\ \ldots\ t_i\ \texttt{*}x_i\ \ldots\ \texttt{) \{}\\ \quad\ldots\\ \quad t_i\ a_i\texttt{;}\\ \quad\ldots\\ \quad f\texttt{(}\ \ldots\ \texttt{\&}a_i\ \ldots\ \texttt{);}\\ \quad\ldots\\ \quad\texttt{*}x_i\ \texttt{=}\ a_i\texttt{;}\\ \quad\ldots\\ \texttt{\}}\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\texttt{void}\ f\texttt{(}\ \ldots\ t_i\ \texttt{*}x_i\ \ldots\ \texttt{) \{}\\ \quad\ldots\\ \quad\ldots\\ \quad f\texttt{(}\ \ldots\ x_i\ \ldots\ \texttt{);}\\ \quad\ldots\\ \quad\ldots\\ \texttt{\}}\end{array}\right]\!\!\right]$$

and

$$\mathbf{TR}\left[\!\!\left[\begin{array}{l}\texttt{void}\ f\texttt{(}\ \ldots\ t_i\ \texttt{*}x_i\ \ldots\ \texttt{) \{}\\ \quad\ldots\\ \quad t_i\ a_i\texttt{;}\\ \quad\ldots\\ \quad f\texttt{(}\ \ldots\ \texttt{\&}a_i\ \ldots\ \texttt{);}\\ \quad\ldots\\ \quad\texttt{*}x_i\ \texttt{= CONS(}exp\texttt{,}\ a_i\texttt{);}\\ \quad\ldots\\ \texttt{\}}\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\texttt{void}\ f\texttt{(}\ \ldots\ t_i\ \texttt{*}x_i\ \ldots\ \texttt{) \{}\\ \quad\ldots\\ \quad\ldots\\ \quad f\texttt{(}\ \ldots\ \texttt{RCONS(}exp\texttt{,}\ x_i\texttt{)}\ \ldots\ \texttt{);}\\ \quad\ldots\\ \quad\ldots\\ \texttt{\}}\end{array}\right]\!\!\right]$$

If by these transformations the recursive call of f becomes tail-recursive, it can be transformed into a loop.

Above transformations are applied on all tuple-generating functions. In the program each call of the original function f is replaced by the call of the corresponding new function g (as shown in the new definition of f).

A further peephole optimization step is applied to replace each application $\texttt{ELEM}_i\texttt{(}z\texttt{)}$ (where z is any variable holding the value $\texttt{TUPLE}_m\texttt{(}z_1\texttt{,}\ \ldots\ \texttt{,}\ z_m\texttt{)}$) by the variable z_i. If all applications of z are of this form, then its definition is removed and the tuple is never generated.

Thus there are only few cases when tuples are actually treated as data structures:

1. A tuple is stored in a data structure.
2. A tuple is returned as the result of an external function.
3. A tuple is passed as a function argument.

Concerning the second case, there is a special form of external function declarations for functions that do not return a tuple but actually expect multiple references to store the function results in (see Section 4.5).

Concerning the third case, we assume that if the programmer explicitly passes a tuple instead of multiple function arguments he actually expects a data structure to be created. However there is one exception to this rule. In an iterated composition

$\texttt{rcomp(}f\texttt{,}\ b\texttt{) [}\ e\ \texttt{|}\ iters\ \texttt{]}$

the base value b becomes an argument of the automatically generated recursive function. Since the only way for a programmer to pass multiple values is to use a tuple for b, the compiler transforms a tuple in this place into multiple function arguments.

decls	$\rightarrow$	`extern type` *id*	External Type
	|	`extern` *id* `::` *type*	External Argument
	|	`extern const` *id* `::` *type*	External Constant
	|	`extern` *id* `::` *funtype*	External Function
	|	`extern proc` *id* `::` *funtype*	External Procedure

Figure 4.6: External Declarations

4.5 The External Interface

In this section, we describe the interaction between the code generated by the compiler and external C types, values, and functions.

4.5.1 The Runtime System

The compiler does not care for the low-level details of data representation and memory management (i.e. garbage collection). The generated code assumes the following types and operations to be implemented by the runtime system:

- *Basic Type:* The atomic type `WORD` is able to hold integers, list references, and tuple references. All types in the D program are mapped to this type in the C program.
- *List Operations:* The functions `CONS`, `RCONS`, `ISNIL`, `HEAD` and `TAIL` are used for list manipulation.
- *Tuple Operations:* The tuple constructors `TUPLE`i, $2 \leq i \leq 6$, and the tuple selectors `ELEM`i, $1 \leq i \leq 6$, manipulate small tuples. For $i > 6$, the list to tuple converter `TUPLE` and the tuple selector `ELEM` are used.

The SACLIB implementation of this interface represents all compound objects as lists; all constructors are boiled down to the `COMP` operation and all selectors down to the functions `FIRST` and `RED`.

Appendix B.3 shows the PACLIB realization of this interface in detail.

4.5.2 The Main Function

The compiler translates an expression in the D language into a file that contains the definition of a single external C function (named `dcmain` by default) which returns the value of the expression. While this file may contain the definitions of other C functions as well, these functions may be only called by `dcmain` or other local functions.

Actually, all generated functions but `dcmain` are declared `static` and are thus only visible within the file. Even without this restriction, there is no guarantee that a function contained in the original D program has a corresponding C counterpart (due to renaming, program transformations and in particular inlining).

In short, `dcmain` is the only function that may be called from other object files. Up to now we have assumed that the functional program is a closed expression without any free variables. In this case the main function does not expect any arguments. The following subsection describes how to make this interface more flexible.

4.5.3 External Types

The compiler allows type names to be declared external. In this case, only the name of the type but not its definition is known to the compiler. Consequently, this type is treated as a basic type like `Int` or `Bool` on which however no basic operations are pre-defined. Type checking of function applications involving these types is performed as usual.

External types are a way to manipulate objects whose structure does not correspond to the type scheme of the D language. For example, the SACLIB user guide says [Buchberger *et al.*, 1992]:

> The SACLIB representation `I` of a number $n \in \mathbf{Z}$ is defined as follows:
>
> - If `-BETA` $< n <$ `BETA` then `I` is the integer whose value is n.
> - If $n \leq$ `-BETA` or `BETA` $\leq n$ then `I` is the list $(d_0, d_1, \ldots, d_k)$ with $d_k \neq 0, d_i \leq 0$ if $n < 0$ and $0 \leq d_i$ if $0 < n$ for $0 \leq i \leq k$ and $n = \sum_{i=0}^{k} d_i \mathtt{BETA}^i$.

This representation does not match any valid D type, consequently we define

```
extern type Integer
```

and do (within a D program) not care for the type's representation.

4.5.4 External Values

The definition of a D program as a closed expression is very restrictive since it allows to implement only *programs* but not *library functions*. A solution to this problem is to define variables as external:

```
extern x :: Integer
extern y :: Bool
```

In this case, the compiler treats these variables as normal constants (whose definitions are however missing) but finally generates a main function that contains the names of these constants as formal parameters:

```
Integer dcmain(Integer x, Bool y) {
  ...
}
```

The order of the parameters reflects the order of the external declarations. This is the only place where the order of definitions in a D program does matter; to avoid errors, external variable declarations should therefore occur only at a single place at the top-level of the program (but this is not checked by the compiler).

There is a second form of external variable definitions:

```
extern const PRIME :: [Integer]
```

A `const` definition is semantically equivalent to an external variable definition, but the compiler generates a different form of target code:

```
extern WORD PRIME;
Integer dcmain(Integer x, Bool y) {
  ...
}
```

Instead of a parameter of the main function, PRIME is a global variable that is given its value in another object file. This form of declarations is mainly useful for variables that are defined once and not changed any more.

4.5.5 External Functions

A D program may also call human-written C functions respectively C functions that have been automatically generated from other D programs. E.g. the declaration

```
extern IPROD :: (Integer, Integer) -> Integer
```

allows the SACLIB function IPROD to be called in a D program. This gives the possibility to reuse existing C source code and also to implement functions whose performance is critical directly in C.

The use of external functions is potentially hazardous in two respects:

1. *Function Interface:* It is entirely the responsibility of the programmer to ensure that the actual function interface matches the declaration in the D program and that the function obeys the calling conventions of D programs. In particular, if the result type of an external function is not an external type, the data representation of the result must match the representation used by the D program.
2. *Referential Transparency:* It is entirely the responsibility of the programmer to ensure that the external function is referentially transparent with respect to the functional layer. The compiler may assume that an external function always returns the same result when given the same input[3].

   Moreover, it is even dangerous, if an external function causes side-effects that are not visible from the functional program (like output): the order in which these side-effects occur is not defined beyond the restrictions imposed by the denotational semantics given in Section 4.1.

There is a special external declaration for functions that return multiple arguments:

```
extern proc IQR :: (Integer, Integer) -> (Integer, Integer)
```

The proc declaration says that IQR is not a function that returns a tuple result but a procedure that expects additionally to the input arguments the addresses of two variables to store the results in:

```
void IQR(Word a, Word b, Word *q, Word *r)
```

This form of declarations is only allowed for tuple results (of at least two components); the compiler optionally generates this form of interface also for the main function of a D program.

4.6 Benchmarks

In this section, we present some benchmark results that demonstrate the efficiency of the generated code.

[3]In Appendix C.1.4 we give the source code of a D program that uses the SACLIB random generator IRAND. However, since l is constant, the compiler is free to replace each call of IRAND(l) by the same constant such that all "random" numbers become equal.

4.6.1 Comparison with SML

First, we give some of the "toy" benchmarks that seem to be standard for comparing implementations of functional languages:

- `primes(`n`)` computes all primes less than n using a simple recursive algorithm without caring for any particular optimizations. Nevertheless the D compiler transforms all recursive functions into loops.
- `queens(`n`)` solves the n-queens problems using two mutually recursive functions. The D compiler transforms the recursion cycle into a single function with two directly recursive calls and transforms one of these recursive calls into loop iteration.
- `qsort(`n`)` sorts a list of n random numbers using the Quicksort algorithm in a version that copes without an explicit append operation. The D compiler transforms one of the recursive calls into loop iteration.

The timings were performed on a DECstation 5000/25Mhz under Ultrix 4.0 where the generated C code was compiled with the GNU C compiler. The results are compared with the New Jersey implementation of Standard ML Release 0.93 which is a modern implementations of a strict functional language [Appel and MacQueen, 1991]. Garbage collection times are excluded.

Benchmark	n	D	SML/NJ 0.93
primes	15000	2580 ms	2550 ms
queens	11	1690 ms	2750 ms
qsort	2500	5350 ms	9150 ms

Basically the D implementation is more than 1.5 times faster than SML/NJ. However, in the `primes` example with comparatively few list constructions but many list accesses, the indirect list referencing scheme of the SACLIB runtime system appears to degrade the performance.

4.6.2 Comparison with SACLIB C

As an example of a more realistic application, we present in Appendix C.1 the full source code of a D program for the exact solution of a linear equation system with integer coefficients. The program is based on a modular algorithm and uses a C subroutine for the computation of determinants by L/U decomposition.

The algorithms applied in the D/C functions are highly optimized. The timings are compared with a program that was entirely written in SACLIB C using the same algorithm (actually the D program was derived from the original SACLIB solution). n is the dimension of the system (i.e. the number of variables) and l the bit-length of its (random) coefficients.

The timings were performed on a Sequent Symmetry 80386/20Mhz under Dynix/ptx 2.0.

Benchmark	n/l	D	SACLIB 1.1
linsolve	5/512	4810 ms	4830 ms
linsolve	10/256	10500 ms	10510 ms
linsolve	40/016	20910 ms	20950 ms

It is not clear to us why the D program is actually slightly *faster* than the original SACLIB solution. One reason may be that the manually coded C program is totally based on SACLIB arrays which are less efficiently implemented than SACLIB lists. Perhaps also the fact that the automatically generated code is very compact makes it more amenable to the optimizations performed by the C compiler.

Chapter 5

The Pseudo-Functional Language

In this chapter, we define the core language that serves as the functional interface to the parallel runtime system. Some features of this language exceed the purely functional framework: task abortion (Subsection 5.5.6), non-determinism (Subsection 5.5.7) and forward tasks (Subsection 5.5.8). However, they lay the basis for the referential transparent abstractions in Section 6.4.5 (cyclic constants) and Section 7.6 (non-deterministic compositions).

This chapter is mainly taken from [Schreiner, 1994b].

5.1 Abstract Syntax

The parallel core language extends the basic core language defined in Figure 4.1 by constructs for the creation and synchronization of tasks and for the composition and decomposition of streams:

1. A *task* decouples the application of a function from the point where the application is defined. Semantically there is no difference whether the function is executed concurrently by a separate task or whether the application is just "frozen" and computed

exp	$\rightarrow$	...see Figure 4.1 ...	Basic Expressions
	|	`start` (id , exp_1 , ... , exp_n)	Task Creation
	|	`wait` (exp)	Task Synchronization
	|	`[ ]s`	Nil Stream
	|	exp_h `:s` exp_t	Stream Construction
	|	`scase` exp `of` $patterns$	Stream Selection
	|	`any` (exp)	Non-Determinism
	|	`stop` exp_t `in` exp_r	Task Abortion
	|	`forward`	Forward Task Creation
	|	`fstart` (f , id , exp_1 , ... , exp_n)	Forward Task Activation

Figure 5.1: Abstract Syntax

when its result is actually required. In other words, the creation of a task is actually a *non-strict function application*.

2. A *stream* differs from standard lists in that the tail field of every stream cell contains a *task* that computes the rest of the stream. This allows the construction of streams that are *non-strict* in their tails and consequently the definition of cyclic streams.

Figure 5.1 gives the abstract syntax of these extensions. Except for `any`, `stop`, `forward`, `fstart` (described at the end of this chapter), the extended semantics fits nicely into the framework described in Section 4.1. We therefore only give the necessary extensions.

5.2 Examples

The following program constructs a list $[f(1), \ldots, f(n)]$ in parallel:

```
list [ wait(y) | y <- l ] where
  l <- list [ start(f,x) | x <- 1..n ]
```

The following program computes the same result using a stream of values:

```
strsync(strmake(f, n)) where
  strmake :: (Int->*a, Int) -> Stream(*a)
  strmake(f,n) =
    if n <= 0
      then snil
      else scons(f(n), start(strmake, f, n-1))
  strsync :: Stream(*a) -> [*a]
  strsync(s) =
    scase s of
      [ ] => [ ]
      h:t => h:strsync(wait(t))
```

The constructors `snil` and `scons` are the actual textual representations of $[\ ]_{\mathtt{S}}$ and $:_{\mathtt{S}}$, respectively. `strsync` transforms a stream into a list by synchronizing with every stream tail. This program becomes much more elegant when using the higher level primitives described in Chapter 7.

5.3 Semantic Algebras

In the following sections, we will extend the valuation functions

$\mathbf{E}:$	Expression $\rightarrow$ ValueEnv $\rightarrow$ Value	Expression Value
$\mathbf{T}:$	Expression $\rightarrow$ TypeEnv $\rightarrow$ Type	Type Value

to handle the new sorts of expressions. Remember that $\mathbf{T}[\![exp]\!]e$ defines the type of exp in context e while $\mathbf{E}[\![exp]\!]e$ defines the result (see Subsection 4.1.3).

The domain 'Value' is extended by

Task =	$Unit \rightarrow$ Value	Task
SNil =	$Unit$	Empty Stream
Stream =	Value $\times$ Value	Non-Empty Stream
Value =	(...see Section 4.1.3...	Denotable Value
	+ Task + SNil + Stream)$_\perp$	

and correspondingly the domain 'Type':

TaskType =	Type	Task
SNilType =	*Unit*	Empty Stream
StreamType =	Type	Non-Empty Stream
Type =	(...see Section 4.1.3...	Denotable Type
	+ TaskType	
	+ SNilType + StreamType)$_\perp$	

The environment domains 'ValueEnv' and 'TypeEnv' are constructed as usual:

ValueEnv =	(Identifier $\rightarrow$ Value)$_\perp$	Value Environment
TypeEnv =	(Identifier $\rightarrow$ Type)$_\perp$	Type Environment

As already mentioned in Section 4.1.3, the construction mk$T(\ldots)$ that embeds a type into some union type is always *strict* in its arguments e.g. mkStream$(\perp, t) = \perp$.

5.4 Typing

The type compatibility function 'eqType' must be extended as follows:

$$
\begin{array}{l}
\text{eqType} : \text{Type} \times \text{Type} \rightarrow \text{Bool} \\
\text{eqType}(T_1, T_2) = \\
\quad T_1 = \text{mkErrorType}() \vee T_2 = \text{mkErrorType}() \rightarrow \text{true}; \\
\quad \text{case } T_1 \text{ of} \\
\qquad \ldots \\
\qquad \text{isTaskType}(t_1) \rightarrow \\
\qquad\quad \text{case } T_2 \text{ of} \\
\qquad\qquad \text{isTaskType}(t_2) \rightarrow \text{eqType}(t_1, t_2) \\
\qquad\qquad \text{otherwise} \rightarrow \text{false} \\
\qquad \text{isSNilType}() \rightarrow \\
\qquad\quad \text{case } T_2 \text{ of} \\
\qquad\qquad \text{isSNilType}() \rightarrow \text{true} \\
\qquad\qquad \text{isStreamType}(t) \rightarrow \text{true} \\
\qquad\qquad \text{otherwise} \rightarrow \text{false} \\
\qquad \text{isStreamType}(t_1) \rightarrow \\
\qquad\quad \text{case } T_2 \text{ of} \\
\qquad\qquad \text{isSNilType}() \rightarrow \text{true} \\
\qquad\qquad \text{isStreamType}(t_2) \rightarrow \text{eqType}(t_1, t_2) \\
\qquad\qquad \text{otherwise} \rightarrow \text{false}
\end{array}
$$

Likewise 'joinType' is extended:

$$
\begin{array}{l}
\text{joinType} : \text{Type} \times \text{Type} \rightarrow \text{Type} \\
\text{joinType}(T_1, T_2) = \\
\quad T_1 = \text{mkErrorType}() \rightarrow T_2; \\
\quad T_2 = \text{mkErrorType}() \rightarrow T_1; \\
\quad \text{case } T_1 \text{ of} \\
\qquad \ldots \\
\qquad \text{isTaskType}(t_1) \rightarrow \\
\qquad\quad \text{case } T_2 \text{ of isTaskType}(t_2) \rightarrow \text{mkTaskType}(\text{joinType}(t_1, t_2)) \\
\qquad \text{isSNilType}() \rightarrow \\
\qquad\quad \text{case } T_2 \text{ of}
\end{array}
$$

$$
\begin{array}{l}
\quad\quad \text{isSNilType}() \rightarrow \ \text{mkSNilType}() \\
\quad\quad \text{isStreamType}(t) \rightarrow \ T_2 \\
\text{isStreamType}(t_1) \rightarrow \\
\quad \text{case } T_2 \text{ of} \\
\quad\quad \text{isSNilType}() \rightarrow \ T_1 \\
\quad\quad \text{isStreamType}(t_2) \rightarrow \ \text{mkType}(\text{joinType}(t_1, t_2))
\end{array}
$$

5.5 Denotational Semantics

5.5.1 Task Creation

$$
\begin{array}{l}
\mathbf{T}[\![\texttt{start(}id\texttt{, } exp_1\texttt{, } \ldots \texttt{ , } exp_n\texttt{)}]\!] = \underline{\lambda} e. \\
\quad \text{case } e([\![id]\!]) \text{ of isFunctionType}_m(t_r, t_1, \ldots\ t_m) \rightarrow \\
\quad\quad (m = n \wedge \forall 1 \leq i \leq n:\ \text{eqType}(t_i, \mathbf{T}[\![exp_i]\!]e)) \rightarrow \text{mkTaskType}(t_r) \\
\mathbf{E}[\![\texttt{start(}id\texttt{, } exp_1\texttt{, } \ldots \texttt{ , } exp_n\texttt{)}]\!] = \underline{\lambda} e. \\
\quad \text{case } e([\![id]\!]) \text{ of isFunction}(f) \rightarrow \\
\quad\quad (\underline{\lambda} a_1 \ldots \underline{\lambda} a_n.\text{mkTask}(\lambda x.f a_1 \ldots a_n))\mathbf{E}[\![exp_1]\!]e\ \ \ldots\ \ \mathbf{E}[\![exp_n]\!]e
\end{array}
$$

The `start` construct is *strict* in the arguments exp_i but not in the result of the function application i.e. `start`$(f,\ \perp) = \perp$ but, if $f(a) = \perp$, then nevertheless `start`$(f,\ a) \neq \perp$. In other words computation may continue without synchronizing with the result of the function application.

5.5.2 Task Synchronization

$$
\begin{array}{l}
\mathbf{T}[\![\texttt{wait(}exp\texttt{)}]\!] = \underline{\lambda} e. \\
\quad \text{case } \mathbf{T}[\![exp]\!]e \text{ of isTaskType}(T) \rightarrow T \\
\mathbf{E}[\![\texttt{wait(}exp\texttt{)}]\!] = \underline{\lambda} e. \\
\quad \text{case } \mathbf{E}[\![exp]\!]e \text{ of isTask}(t) \rightarrow t()
\end{array}
$$

The `wait` brings a task "down-to-earth" by synchronizing with its result; in particular we have `wait`$(\text{mkTask}(\lambda x.\perp)) = \perp$

5.5.3 Empty Stream

$$
\begin{array}{l}
\mathbf{T}[\![\texttt{[ ]}_\texttt{S}]\!] = \underline{\lambda} e.\text{mkSNilType}() \\
\mathbf{E}[\![\texttt{[ ]}_\texttt{S}]\!] = \underline{\lambda} e.\text{mkSNil}()
\end{array}
$$

5.5.4 Stream Construction

$$
\begin{array}{l}
\mathbf{T}[\![exp_h\ \texttt{:}_\texttt{S}\ exp_t]\!] = \underline{\lambda} e. \\
\quad \text{case } \mathbf{T}[\![exp_t]\!]e \text{ of isTaskType}(t_t) \rightarrow \\
\quad\quad \text{joinType}(t_t, \text{mkStreamType}(\mathbf{T}[\![exp_h]\!]e)) \\
\mathbf{E}[\![exp_h\ \texttt{:}_\texttt{S}\ exp_t]\!] = \underline{\lambda} e. \\
\quad \text{mkStream}(\mathbf{E}[\![exp_h]\!]e, \mathbf{E}[\![exp_t]\!]e)
\end{array}
$$

Stream construction differs from list construction in that the tail exp_t is actually a task that computes the rest of the stream. However, the construction of the stream cell is still strict; only in the case of an application `(`$\ldots$ `:`$_\texttt{S}$ `start(`$\ldots$`))` with all arguments of `start` being simple variables the tail is constructed in a non-strict fashion.

5.5.5 Stream Selection

$\mathbf{T}[\![$`scase` exp_s `of [ ] =>` exp_n `;` id_h `:` id_t `=>` $exp_c]\!] = \underline{\lambda} e.$
case $\mathbf{T}[\![exp_s]\!]e$ of
isErrorType() $\rightarrow$ joinType($\mathbf{T}[\![exp_n]\!]e, \mathbf{T}[\![exp_c]\!]e$);
isSNilType() $\rightarrow$ joinType($\mathbf{T}[\![exp_n]\!]e, \mathbf{T}[\![exp_c]\!]e$);
isStreamType(T) $\rightarrow$
joinType($\mathbf{T}[\![exp_n]\!]e$,
$\mathbf{T}[\![exp_c]\!][[\![id_h]\!] \leftarrow T, [\![id_t]\!] \leftarrow$ mkTaskType(mkStreamType(T))] e

As for the `case` construct (Subsection 4.1.9), `scase` cannot give types for id_h and id_t if exp_s is the `error` constant. The variables must not be used then.

$\mathbf{E}[\![$`scase` exp_s `of [ ] =>` exp_n `;` id_h `:` id_t `=>` $exp_c]\!] = \underline{\lambda} e.$
case ($\mathbf{E}[\![exp_s]\!]e$) of
isSNil() $\rightarrow \mathbf{E}[\![exp_n]\!]e$;
isStream(h, t) $\rightarrow$ $\mathbf{E}[\![exp_c]\!][[\![id_h]\!] \leftarrow h, [\![id_t]\!] \leftarrow t]$ e

Please note that id_t is still a task; the `scase` construct only decomposes a stream cell but does *not* synchronize for the stream tail. For this purpose, an additional application of `wait` is required.

5.5.6 Task Abortion

Informally the purpose of the expression

`stop` *tasks* `in` *exp*

is to assert that the result values of the denoted *tasks* are not required any more and that consequently the *tasks* may be terminated before *exp* is evaluated. Unlike the other expressions, the effect of `stop` is an *imperative* one; simply replacing it by *exp* would not change the denotational semantics of the program but only affect its performance.

However, this is also the reason why `stop` may be applied in a restricted context only. The denoted *tasks* must not be in use anywhere else in the program because otherwise some `wait` (or `any`) may block forever (depending on the runtime situation).

The imperative nature of `stop` does not well fit into the purely denotational language framework. One solution would be to resort to the notion of an imperative *storage* in which tasks reside and that is referenced by the task handles accessible in the program. However, consequently the semantics of the program would have to be defined by all possible interleavings of the evaluations of `stop` and `wait` expressions; it would become an *imperative* one and the beauty and simplicity of the functional framework would be lost.

We take another approach that captures the essence of the `stop` construct by asserting that it is only applied in the restricted context described above:

1. First the program is transformed into *linear form* [Wadler, 1991] where every identifier has at most one applied occurrence. This is accomplished by introducing auxiliary `copy` constructs that duplicate their argument.
2. Then we define a denotational semantics that keeps track of whether an object is only referenced once or not (simulating the effect of *reference counting*).
3. The `stop` construct is then able to assert that the denoted tasks are not in use anywhere else and may safely change their denotation.

The program denotation is now defined as

$$\mathbf{P}[\![exp]\!] = \text{let } (c, v) = \lambda e.\mathbf{EA}[\![\ \mathbf{TA}[\![exp]\!]\ ]\!] e \text{ in } v$$

where the interface of the new evaluation function **EA** is

$$\mathbf{EA} : \text{Expression} \rightarrow \text{CValueEnv} \rightarrow \text{CValue}$$

and the new semantic domains

$$\begin{array}{lcl} \text{CValue} & = & \mathbb{N} \times \text{ Value} \\ \text{CValueEnv} & = & \text{Ident } \rightarrow \text{ CValue} \end{array}$$

are introduced. 'CValue' is a pair of an integer and a value where the integer denotes how often the value is referenced in the program.

The transformation of the program into linear form is performed by repeated applications of the rule

$$\mathbf{TA}\left[\!\!\left[\begin{array}{l} \dots \\ x \texttt{ = } \dots \\ \dots \\ \dots\ x\ \dots\ x\ \dots \\ \dots \end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l} \dots \\ x \texttt{ = } \dots \\ \texttt{(}x_1\texttt{, } x_2\texttt{) = copy(}x\texttt{)} \\ \dots \\ \dots\ x_1\ \dots\ x_2\ \dots \\ \dots \end{array}\right]\!\!\right]$$

The definition of constants (Subsection 4.1.7) has to be extended by

$$\begin{array}{l} \mathbf{DA}[\![id_1^{T_1} \texttt{ = } exp_1 \texttt{ ; } \dots \texttt{; (}id_a\texttt{, } id_b\texttt{) = copy(}exp\texttt{); } \dots \texttt{ ; } id_n^{T_n} \texttt{ = } exp_n]\!] = \underline{\lambda} e_0. \\ \quad \text{let } e_1 = [[\![id_1]\!] \leftarrow \mathbf{EA}[\![exp_1]\!] e_0]\ e_0 \text{ in} \\ \quad \dots \\ \quad \text{let } e_{i+1} = [[\![id_a]\!] \leftarrow \mathbf{EA}[\![\texttt{copy(}exp\texttt{)}]\!] e_i]\ e_i \text{ in} \\ \quad \text{let } e_{i+2} = [[\![id_b]\!] \leftarrow \mathbf{EA}[\![\texttt{copy(}exp\texttt{)}]\!] e_i]\ e_{i+1} \text{ in} \\ \quad \dots \\ \quad \text{let } e_n = [[\![id_n]\!] \leftarrow \mathbf{EA}[\![exp_n]\!] e_{n-1}]\ e_{n-1} \text{ in} \\ \qquad e_n \end{array}$$

where

$$\mathbf{EA}[\![\texttt{copy(}exp\texttt{)}]\!] = \lambda e.\ \text{let } (n, v) = \mathbf{EA}[\![exp]\!] e \text{ in } (n+1, v)$$

In all other sorts of expressions, the valuation **EA** resembles that of **E** except that the reference count is ignored for the arguments of strict operations and new data structures are created with an initial reference count of 1, e.g.

$$\begin{array}{l} \mathbf{EA}[\![exp_h \texttt{ : } exp_t]\!] = \lambda e. \\ \quad (1, \text{mkList}(\mathbf{EA}[\![exp_h]\!] e, \mathbf{EA}[\![exp_t]\!])) \end{array}$$

We omit these details. The denotation of `stop` finally becomes

$$\begin{array}{l} \mathbf{T}[\![\texttt{stop } exp_t \texttt{ in } exp_r]\!] \ = \ \lambda e. \\ \quad \text{case } \mathbf{T}[\![exp_t]\!] e \text{ of isListType}(T) \rightarrow \\ \quad \text{case } T \text{ of isTaskType}(S) \rightarrow \\ \qquad \mathbf{T}[\![exp_r]\!] e \end{array}$$

$$\begin{array}{l} \mathbf{EA}[\![\texttt{stop } exp_t \texttt{ in } exp_r]\!] \ = \ \lambda e. \\ \quad \text{let} \\ \qquad (n, t) = \ \mathbf{EA}[\![exp_t]\!] e \\ \qquad c = \ \max(n, \text{maxcnt}(t)) \\ \quad \text{in } (c = 1) \rightarrow \mathbf{EA}[\![exp_r]\!] e \end{array}$$

where

$$
\begin{array}{l}
\text{maxcnt} : \text{List} \rightarrow \mathbb{N} \\
\text{maxcnt}(l) = \\
\quad \text{case } l \text{ of} \\
\quad\quad \text{isNil}() \rightarrow 0 \\
\quad\quad \text{isList}(h, t) \rightarrow \\
\quad\quad\quad \text{let } (h_c, h_v) = h; (t_c, t_v) = t \\
\quad\quad\quad \text{in } \max\{h_c, t_c, \ \text{maxcnt}(t_v)\}
\end{array}
$$

In other words, neither the task list itself nor any of its elements must be referenced more than once in a legal application of `stop`; the result is $\perp$ otherwise.

Having formally defined this restriction, we will not use **EA** any more but resort to the simple valuation **E**.

5.5.7 Non-Determinism

The introduction of the non-deterministic construct **any** requires a fundamental change in the denotational semantics of the language. Informally, **any**(*tlist*) may return the return value of any *task* in *tlist* (actually a tuple whose first element is *task* and whose second element is *tlist* without *task*).

By this construct a program expression may return not only a single determinate value but one from a whole *set* of possible values. Even if the outermost operator is deterministic, any subexpression may contain a call of **any** and the whole expression thus becomes non-deterministic. We will discuss the pragmatics of non-deterministim later; let's now concentrate on the formal aspects.

In the non-deterministic language the valuation **E** is replaced by the valuation

$$\mathbf{ES} : \quad \text{Expression} \rightarrow \mathbb{P}(\text{ValueEnv}) \rightarrow \mathbb{P}(\text{Value}) \quad \text{Value Set}$$

that returns a set of return values ($\mathbb{P}$ is the powerset constructor). Moreover, it also takes a set of possible value environments since the expression may now be evaluated in many possible contexts. The semantics of a program (see Subsection 4.1.5) is then defined as

$$\mathbf{P}[\![\texttt{let}\ \mathit{fdefs}\ \texttt{in}\ \mathit{exp}]\!] = \mathbf{ES}[\![\texttt{let}\ \mathit{fdefs}\ \texttt{in}\ \mathit{exp}]\!]\{[\,]\}$$

denoting that the result may be *any* of the values in the resulting denotation set. If this set is empty (this may only happen if **any** is applied to an empty task list) the program is deadlocked.

On the other hand the denotation **T** remains the same since the static type of an expression is the same for all possible evaluations. The type of **any** is therefore easily defined as:

$$
\begin{array}{l}
\mathbf{T}[\![\texttt{any(}\mathit{exp}\texttt{)}]\!] = \underline{\lambda} e. \\
\quad \text{case } \mathbf{T}[\![\mathit{exp}]\!]e \text{ of isListType}(T) \rightarrow \text{case } T \text{ of isTaskType}(S) \rightarrow \\
\quad\quad \text{mkTupleType}(S, \text{mkListType}(\text{mkTaskType}(S)))
\end{array}
$$

The valuation function however becomes more complicated:

$$\mathbf{ES}[\![\texttt{any(}\mathit{exp}\texttt{)}]\!] = \underline{\lambda} E. \bigcup\{\text{any}(t) | e \in E, t \in \mathbf{ES}[\![\mathit{exp}]\!]e\}$$

The function takes a set of environments E and, for any $e \in E$ and for any task list to which *exp* evaluates in e, a result is constructed. The valuation is then the union of possible results.

The auxiliary function

$$
\begin{array}{l}
\text{any} : \text{List} \rightarrow \mathbb{P}(\text{Tuple}) \\
\text{any}(t) = \\
\quad \text{let } (n, f) = \text{seq}(t) \\
\quad \text{in } \{\text{mkTuple}_2(s(0), \text{list}(s, 1, n)) \mid \exists s : \mathbb{N}^n \rightarrow \text{Value}, \exists p : \mathbb{N}^n \xrightarrow{bij.} \mathbb{N}^n, s(i) = f(p(i))\}
\end{array}
$$

shows that each result is a tuple consisting of the result of one task and a list that contains all other tasks. This tuple is constructed in three steps:

1. First 'seq(t)' transforms the task list t into a function f that represents a mathematical sequence i.e. a mapping $\mathbb{N}^n \rightarrow$ Value (where $\mathbb{N}^n = \{0, \ldots, n-1\}$).
2. Then we apply all possible permutations p to f yielding all possible permutated sequences s.
3. From s we take the first element as the result value; the rest sequence is converted back into a list using the function 'list'.

The definitions of the auxiliary functions are given below:

$$
\begin{array}{l}
\text{seq} : \text{List} \rightarrow \mathbb{N} \times (\mathbb{N} \rightarrow \text{Value}) \\
\text{seq}(l) = \\
\quad \text{case } l \text{ of} \\
\quad\quad \text{isNil}() \rightarrow (0, \lambda x.\bot) \\
\quad\quad \text{isList}(h, t) \rightarrow \\
\quad\quad\quad \text{let } (n, f) = \text{seq}(t) \\
\quad\quad\quad \text{in } (n+1, \lambda x.(x = n) \rightarrow h; f(x))
\end{array}
$$

$$
\begin{array}{l}
\text{list} : (\mathbb{N} \rightarrow \text{Value}) \times \mathbb{N} \times \mathbb{N} \rightarrow \text{List} \\
\text{list}(f, i, n) = (i > n) \rightarrow \text{mkNil}();\ \text{mkList}(f(i), \text{list}(f, i+1, n))
\end{array}
$$

Please note that the result list (the second element of the returned tuple) need not contain the rest tasks in the same order as the input list of tasks; actually both input list and result list are actually treated as *multi-sets* (sets that may contain the same element more than once).

What remains now is to redefine the semantics of all other expressions embedding them into the non-deterministic framework. E.g. from the original valuation **E** of the binary operator +

$$
\begin{array}{l}
\mathbf{E}[\![exp_1 \texttt{ + } exp_2]\!] = \underline{\lambda} e. \\
\quad \text{case } \mathbf{E}[\![exp_1]\!] e \text{ of isInt}(i_1) \rightarrow \\
\quad \text{case } \mathbf{E}[\![exp_2]\!] e \text{ of isInt}(i_2) \rightarrow \\
\quad\quad \text{mkInt}(i_1 + i_2)
\end{array}
$$

we get the embedded non-deterministic valuation **ES**

$$
\begin{array}{l}
\mathbf{ES}[\![exp_1 \texttt{ + } exp_2]\!] = \underline{\lambda} E. \\
\quad \bigcup \{\text{sum}(v_1, v_2) | \exists e \in E, v_1 \in \mathbf{ES}[\![exp_1]\!] e, v_2 \in \mathbf{ES}[\![exp_2]\!] e\}
\end{array}
$$

where

$$
\begin{array}{l}
\text{sum} : (\text{Value} \times \text{Value}) \rightarrow \text{Value} \\
\text{sum}(v_1, v_2) = \\
\quad \text{case } v_1 \text{ of isInt}(i_1) \rightarrow \\
\quad \text{case } v_2 \text{ of isInt}(i_2) \rightarrow \\
\quad\quad \text{mkInt}(i_1 + i_2)
\end{array}
$$

For the other sorts of expressions, the transformation is basically the same. In the body of the definition all applications of **E** to subexpressions are abstracted into variables whose values are taken from all possible denotations of the subexpressions in all possible environments passed to the valuation. We therefore omit the detailed definitions.

Within the non-deterministic framework, functional constructs are separated from non-deterministic constructs by the following lemma ($|S|$ denotes the cardinality of a set S):

Definition 12 (Determinism) *An expression exp is* deterministic*, if*

$$|\mathbf{ES}[\![exp]\!] E| = 1$$

for all environment sets E with $|E| = 1$.

We then obviously have

Lemma 1 (Any) *The expression* $[\![\texttt{any}(exp)]\!]$ *is not deterministic.*

but also, by above construction,

Lemma 2 (Embedding) *All sorts of expressions defined in Section 4.1 are deterministic when embedded into the non-deterministic framework.*

We will apply Definition 12 in Section 7.6 to a more interesting situation.

5.5.8 Forward-Defined Tasks

Cyclic constant definitions have proved to be a very useful concept in non-strict functional languages. For instance, one may write the program

```
fib where
  fib = 1:1:f(fib)

  f(fib) =
    case fib of
     [ ] => Error("fib")
     a:l =>
       case l of
         [ ] => Error("fib")
         b:r => (a+b):f(r)
```

constructing the list of Fibonacci numbers.

However, in the functional core language, cyclic definitions are forbidden (Subsection 4.1.7). The order of definitions does not matter, but there has to exist some topological sorting by which data dependencies can be resolved. Anyway, cyclic constant definitions do not make sense because all expressions are evaluated in a strict fashion.

By the introduction of tasks (and consequently streams), the situation has changed; in Subsection 6.4.5 we will describe the possibility of constructing cyclic constant definitions. Because of the strict semantics of our language, such definitions are restricted to constants with task-types. Above program will then be written as

```
fib where
  para fib :: Task(Stream(Int))
  fib = 1:1:f(fib)
```

```
f(fib) =
  case fib of
   [ ] => Error("fib")
   a:l =>
     case l of
       [ ] => Error("fib")
       b:r => (a+b):f(r)
```

from which the compiler generates the following program in the core language

```
strsync(wait(fib)) where
  fib0 :: Task(Stream(Int))
  fib0 = forward;
  fib = fstart(fib0, f0, fib0)

  f0(fib) = scons(1, start(f1, fib))
  f1(fib) = scons(1, start(f2, fib))

  f2(fib) =
   scase wait(fib) of
     [ ] => Error("fib")
     a:l =>
       scase wait(l) of
         [ ] => Error("fib")
         b:r => scons(a+b, start(f2, r))
```

Here a task `fib` is created that executes `f0(fib0)`. `fib0` is a `forward` declared task handle; the `fstart` call returns this `fib0` as the task `fib`.

Consequently, cyclic task definitions are compiled into pairs of `forward`/`fstart` calls. Thus the semantics of this construction is defined by resorting to the original construction:

$$\mathbf{E}[\![\texttt{let}\ \mathit{cdefs}\ \texttt{in}\ \mathit{exp}]\!] = \underline{\lambda} e.\mathbf{E}[\![\mathit{exp}]\!](\mathbf{DE}(\mathbf{FD}[\![\mathit{cdefs}]\!])e)$$

$$\mathbf{FD}\left[\!\!\left[\begin{array}{l} \ldots \\ x\ \texttt{= forward;} \\ y\ \texttt{= fstart(}x\texttt{,}\ f\texttt{,}\ x\texttt{, ...);} \\ \ldots \\ \boxed{\ldots y \ldots} \end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l} \ldots \\ y\ \texttt{= start(}f\texttt{,}\ y\texttt{, ...);} \\ \ldots \\ \boxed{\ldots y \ldots} \end{array}\right]\!\!\right]$$

1. The `forward` definition is removed and every occurrence of the identifier is replaced by the return value of the corresponding `fstart`.
2. The `fstart` call is replaced by a conventional `start` call where the first argument (the forward identifier) is removed.

The semantics of the resulting cyclic definition is described by fixpoint iteration (similar to recursive function definitions).

$$\begin{array}{l} \mathbf{DE}[\![id_1^{T_1}\ \texttt{=}\ exp_1\ \texttt{;}\ \ldots\ \texttt{;}\ id_n^{T_n}\ \texttt{=}\ exp_n]\!] = \underline{\lambda} e. \\ \quad \mathrm{fix}(\lambda e'. \\ \qquad [[\![id_1]\!] \leftarrow \mathbf{E}[\![exp_1]\!]e' \\ \qquad \ldots \\ \qquad [\![id_n]\!] \leftarrow \mathbf{E}[\![exp_n]\!]e']e) \end{array}$$

5.6 Code Generation

The generation of C code for the parallel extensions of the pseudo-functional language is rather straightforward:

1. Tasks are represented by pointers to result structures as described in Section 2.4; task variables have the new type `TASK` (mapped by the external interface to the standard PACLIB type `Word`).

2. Streams are represented by conventional list cells; the empty stream is denoted by the constant `NIL`. Consequently, the core language operations $\mathtt{[\ ]}_\mathtt{S}$ and $\mathtt{:}_\mathtt{S}$ are mapped to `NIL` and `CONS`; the `scase` construct is (like `case`) compiled into the lower level operations `ISNIL`, `HEAD` and `TAIL`.

3. The translation of the basic parallel expression is

$$\mathbf{DC}\ [[\mathtt{start(}f\mathtt{,}\ a_1\mathtt{,}\ \ldots\mathtt{,}\ a_n\mathtt{)}]] = [[\mathtt{START}n\mathtt{(}f\mathtt{,}\ a_1\mathtt{,}\ \ldots\mathtt{,}\ a_n\mathtt{)}]]$$

$$\mathbf{DC}\ [[\mathtt{wait(}\mathit{task}\mathtt{)}]] = [[\mathtt{WAIT(}\mathit{task}\mathtt{)}]]$$

$$\mathbf{DC}\ [[\mathtt{any(}\mathit{tlist}\mathtt{)}]] = [[\mathtt{ANY(}\mathit{tlist}\mathtt{)}]]$$

4. The compilation of the `stop` expression involves the generation of a statement that does not return a value but produces a side-effect (similar to the `ERROR` statement, see Subsection 4.4.1):

$$\mathbf{DC}\ \left[\left[\ \boxed{\begin{array}{l}\ddots \\ \qquad \mathtt{stop}\ \mathit{tlist}\ \mathtt{in}\ \mathit{exp}\end{array}}\ \right]\right] = \left[\left[\ \begin{array}{l}\mathtt{STOP(}\mathit{tlist}\mathtt{);} \\ \boxed{\begin{array}{l}\ddots \\ \qquad \mathit{exp}\end{array}}\end{array}\ \right]\right]$$

The `STOP` statement is embedded into the closest surrounding statement sequence generated by the compiler (e.g. the body of the enclosing function definition is compiled into such a sequence).

5. Pairs of `forward/fstart` definitions are translated as

$$\mathbf{DC}\ [[\mathtt{fstart(}t\mathtt{,}\ f\mathtt{,}\ a_1\mathtt{,}\ \ldots\mathtt{,}\ a_n\mathtt{)}]] = [[\mathtt{FSTART}n\mathtt{(}t\mathtt{,}\ f\mathtt{,}\ a_1\mathtt{,}\ \ldots\mathtt{,}\ a_n\mathtt{)}]]$$

$$\mathbf{DC}\ [[\mathtt{forward}]] = [[\mathtt{FORWARD}]]$$

The external runtime interface described in Appendix B.3 maps `START`, `WAIT`, `ANY`, `STOP`, `FORWARD`, `FSTART` (with minor modifications) into the corresponding PACLIB operations `pacVirtual`, `pacWaitOne`, `pacWaitList`, `pacStopList`, `pacForward` and `pacVirtualFwd`.

The virtual task management scheme is utilized by default (Section 2.6). By a compiler switch, the programmer may explicitly demand eager task creation.

Chapter 6

The Para-Functional Language

In this chapter, we define the parallel annotation language by which the programmer may trigger parallel execution without caring for low level details such as the data type `Task` and the corresponding conversion operations. The compiler translates para-annotated programs into the pseudo-functional language of Chapter 5.

This chapter is mainly taken from [Schreiner, 1994b].

6.1 Motivation

The parallel language defined in Section 5 serves as a interface between the pure functional language and the parallel runtime system. For this purpose, it introduces the type `Task` and corresponding operations `start` and `wait`. However, the level of abstraction provided by these primitives is rather low.

A minor inconvenience is that only function applications (but not general expressions) may be transformed into tasks. A more fundamental problem results from the strong type system of the language. This type system is essential since it allows the compiler to generate efficient target code without runtime type checks.

However this type system also makes the transformation of a functional program in a parallel one a cumbersome task that requires much mental overhead. Introducing `start` calls transforms objects into tasks that may be stored in other data structures and returned by or passed to other functions. The programmer has to trace the type changes through the program, create new functions with modified interfaces, and finally insert `wait` calls at the required places.

One possibility to overcome this problem is to restrict all changes to the body of a function by disallowing tasks (or structures containing tasks) to be returned as function results or passed as function arguments. However then we would loose potential parallelism.

Another solution is to make tasks transparent to the target code and insert runtime type checks at all strict (in particular arithmetic) operations such that tasks are synchronized automatically. However this would significantly slow down the program and trigger more synchronization calls than actually necessary.

In this section we devise a third scheme where the programmer uses type-transparent annotations to trigger parallel computation. From these annotations the compiler generates

exp	$\rightarrow$	...see Figure 4.1 ...	Basic Expressions
	$\mid$	*exp*`@`	Task Annotation
	$\mid$	exp_h `:@` exp_t	Stream Annotation
	$\mid$	*exp*`!`	Synchronization Annotation
defs	$\rightarrow$	...see Figure 4.1 ...	Basic Definitions
	$\mid$	`para` *id* `::` *type*	Para-Type Signature

Figure 6.1: Para-Functional Annotations

the necessary `start` calls, analyzes the program, and inserts the required `wait` invocations. The goal to maximize exploitable parallelism but minimize the synchronization overhead. The resulting program is strongly typed and can be compiled without runtime type checks.

A central component of the compiler's typing algorithm is to extend an expression $e : T$ to an expression $e' : T'$ where T' is a type "larger" than T with respect `Task` constructions. This has yielded the idea to extend the para-functional interface by type annotations e.g.

```
para l :: [Task(T)]
l = ...
```

If the programmer annotates expressions in this way, the compiler will without further assistance generate the required constructs for task creation and synchronization in reasonable places. The resulting programming style yields to a new paradigm of "parallel programming by type annotations" that has to our knowledge not been investigated before.

6.2 The Annotation Language

In Figure 6.1 we show the set of annotations by which the programmer may trigger the parallel evaluation of expressions. The fundamental difference to the constructs of the parallel core language is that the annotations are ignored during the initial phases of compilation, in particular during type checking. In the parallelization phase, however, the compiler will use these hints given by the programmer to generate appropriate parallelization and synchronization constructs, to adapt the types of expressions and to generate new functions.

From the semantic point of view, the annotations can be basically ignored; any functional program will produce the same result after annotations have been inserted. In this sense, the semantics of the annotated language is a superset of the sequential core language. The converse, however, is not true: some annotated programs that properly compile and execute will fail to compile or not terminate when the annotations are removed.

The reason is that the parallelization annotations enrich the expressive power of the language by *non-strict* computations. In the core language (without annotations) certain constructs can therefore not be used (cyclic constant definitions) while others have a different termination semantics (infinite subcomputations).

6.3 Examples

From the following annotated program

```
sumsq(l) where
  l = list [ g(f(x), y)@ | x <- 1 .. n | y <- 1 .. x ]

  sumsq :: [Int] -> [Int]
  sumsq(l) = rcomp(sum,0) [ x*x | x <- l ]
```

the compiler will generate

```
sumsq(l) where
  l = list [ start(h, x, y) | x <- 1 .. n | y <- 1 .. x ]

  h :: (Int, Int) -> Int
  h(x,y) = g(f(x), y)

  sumsq :: [Task(Int)] -> [Int]
  sumsq(l) = rcomp(sum,0) [ y*y | x <- l, y = wait(x) ]
```

where `sumsq` runs concurrently with the construction of the list. The same effect may be also achieved by a type annotation

```
sumsq(l) where
  para l :: [Task(Int)]
  l = list [ g(f(x), y) | x <- 1 .. n | y <- 1 .. x ] )

  sumsq :: [Int] -> [Int]
  sumsq(l) = rcomp(sum,0) [ x*x | x <- l ]
```

The following program

```
primes :: Int -> [Int]
primes(n) =
  if n < 3
    then Nil
    else plist where
      para plist :: Stream(Int)
      plist = 3 : sieve(n, 5, plist)

sieve :: (Int, Int, [Int]) -> [Int]
sieve (n, t, plist) =
  if t > n
    then Nil
    else test(t, plist, rest) where
      rest = sieve(n, t+2, plist)@

test :: (Int, [Int], [Int]) -> [Int]
test (t, pplist, rest) =
  case pplist of
    Nil     => Nil
    p:plist =>
      if t < p*p
        then t : rest
        else if t % p == 0
          then rest
          else test(t, plist, rest)
```

is compiled into a parallel program that creates a list of primes using a recursively defined stream; each prime number in this stream is computed by a parallel `sieve` task.

6.4 Denotational Semantics

In the following paragraphs we give a denotational description of the annotations. This description is partially informal but tries to capture the additional expressive power (and to some extent also the changed operational behavior) of the language. However the actual definition of the translation process requires more background which is given in the Section 6.5.

The following definitions use the function

$$
\begin{array}{l}
\text{sync} : \text{ Value } \rightarrow \text{ Value} \\
\text{sync}(e) = \\
\quad \text{case } e \text{ of} \\
\qquad \text{isBool}(b) \rightarrow b \\
\qquad \text{isInt}(i) \rightarrow i \\
\qquad \text{isList}(h, t) \rightarrow \text{mkList}(\text{sync}(h),\ \text{sync}(t)) \\
\qquad \ldots \\
\qquad \text{isTask}(t) \rightarrow \text{sync}(t()) \\
\qquad \text{isStream}(h, t) \rightarrow \text{mkList}(\text{sync}(h), \text{sync}(t))
\end{array}
$$

that synchronizes any 'Value' into a ground object that does not contain any 'Task' (and consequently also not any 'Stream' object).

6.4.1 Task Annotations

$$\mathbf{TE}[\![\mathit{exp}\texttt{@}]\!] = \underline{\lambda} e.\text{mkTask}(\underline{\lambda} v.\mathbf{TE}[\![\mathit{exp}]\!] e)$$

The annotation denotes that the evaluation of *exp* is decoupled from the current task (e.g. by having *exp* be evaluated by a concurrent task); in other words, evaluation is performed in a *non-strict* fashion.

Since the static type of *exp* remains unchanged, we have to extend the definitions of all strict operators in the core language by a case that checks whether its argument is a task or not, e.g.

$$\mathbf{TE}[\![\hat{}\, \mathit{exp}]\!] = \underline{\lambda} e.\text{sync}(\mathbf{TE}[\![\mathit{exp}]\!] e)$$

Tasks are therefore only synchronized at the strict basis operations.

This definition reflects a common implementation of concurrent tasks in functional languages by adding a tag that is checked on every strict operator. Please note, however, that this is not true for our implementation where all objects are statically typed and the compiler will insert the necessary synchronizations at *compile time*.

6.4.2 Stream Annotations

$$
\begin{array}{l}
\mathbf{TE}[\![\mathit{exp}_h \texttt{:@}\ \mathit{exp}_t]\!] = \underline{\lambda} e.\text{mkStream}(a, b) \text{ where} \\
\quad a = \mathit{adapt}[T](\mathbf{TE}[\![\mathit{exp}_h]\!] e) \\
\quad b = \text{mkTask}(\underline{\lambda} v.\mathit{adapt}[\text{Stream}(T)](\mathbf{TE}[\![\mathit{exp}_t]\!] e))
\end{array}
$$

exp_t is evaluated by a concurrent task whose result is a stream. The result of the whole construct therefore is a stream cell whose tail is evaluated in a non-strict fashion. Please note that this does not yet guarantee that the tails of the cells in exp_t are constructed in a non-strict fashion as well.

A major complication arises from the fact that in a strongly typed language all elements of a stream have to be of the same type. This is not true any more in the presence of annotations since some elements may have been annotated differently than others.

This constraint is neglected by the sequential type checker (that ignores parallelization annotations) but has to be considered by the parallelization phase. The pseudo-function *adapt* indicates that it is necessary to transform the head and the tail elements to some common base type T.

6.4.3 Synchronization Annotations

$$\mathbf{TE}[\![exp\,\texttt{!}]\!] = \underline{\lambda} e.\mathrm{sync}(\mathbf{TE}[\![exp\,\texttt{!}]\!] e)$$

The annotation guarantees that the result does not contain any tasks or streams by synchronizing *exp* to ground objects. It may be used to improve the synchronization decisions made by the compiler.

6.4.4 Para-Type Signatures

$$\begin{array}{l}
\mathbf{DE}[\![\ldots\ \texttt{para}\ id\ \texttt{::}\ type\texttt{;}\ id\ \texttt{=}\ exp\ \ \ldots]\!] = \underline{\lambda} e. \\
\quad \mathrm{fix}(\lambda e'. \\
\qquad [\ldots \\
\qquad\quad [\![id]\!] \leftarrow extend[type](\mathbf{E}[\![exp]\!] e') \\
\qquad \ldots] e)
\end{array}$$

The purpose of the para-functional type signature is to have the definition of the denoted constant value be transformed such that its value is extended to the denoted type. It is an error if the para-functional type of *exp* is already larger or not comparable with *type* (see the next section).

A subtle issue arises if $\mathbf{E}[\![exp]\!] e = \perp$ (ignoring all annotations in *exp*). It may be then the case that the annotation causes the compiler to transform a non-terminating subcomputation in *exp* into a task. Thus the program may finally return a value $v \neq \perp$ which would be the result of the program under a lazy evaluation strategy (a similar situation arose for the constant folding optimizations described in Section 4.3.5).

6.4.5 Cyclic Constant Definitions

The para-annotated language allows to write cyclic constant definitions as described in Subsection 5.5.8. Since the compiler applies a simplified typing algorithm, the programmer must give for such constants explicit `para` annotations, e.g.

```
para fib :: Task(Stream(Int))
fib = 1:1:f(fib)
```

The semantics of cyclic definitions is described in Subsection 5.5.8 by means of fixpoint iteration. However, this semantics is actually only established by the compiler if the following conditions are satisfied:

1. Every non-`Task` constructor in the type of the cyclic constant is annotated by a `Task` constructor e.g.

   ```
   Task([Task(Int)])
   Task((Task(Int),Task(Bool)))
   ```

   are valid types for a recursive constant but

   ```
   Task([Int])
   ```

   is not (neither is actually the type of the `fib` constant above).

2. Every expression constructing a component of the result object is appropriately annotated by `@` respectively `:@`. E.g. the expression

   ```
   para x :: Task((Task(Int),Task(Int)))
   x = (1@,(1+x.0)@)@
   ```

   is appropriately annotated yielding a tuple (1,2). Dynamically, the task computing this tuple starts two subtasks and returns a tuple of tasks. The first task returns 1; the second task waits for the result of the parent task and for the result of the sister task stored in `x.0`; this value will be used to compute the second component.

   Consequently, the following program

   ```
   para x :: Task((Task(Int),Task(Int)))
   x = (1@,1+x.0@)@
   ```

   is not valid since the compiler generates from this program a main task that has to wait for the result of the second subtask in order to add 1 and define the second tuple component. This code leads to a deadlock.

However, in typical situations these restrictions may be considerably relaxed:

If the programmer knows that non-strict evaluation is only required for specific components of a recursively defined data structure, only these components have to be tasks. For instance, for the recursive construction of the Fibonacci stream, it is not necessary that the individual list elements are tasks; consequently the type `Task(Stream(Int))` is sufficient (as well as the type `Task(Int,Task(Int))` for the tuple construction shown above).

Furthermore, it is usually not necessary for the programmer to provide explicit `@` annotations. The result extension algorithm described in Subsection 6.10 constructs from the `para` declaration these annotations in the same way as the programmer would do. Only in case the algorithm is not able to eliminate $ext[T, T']$ functions (and the compiler gives a warning), the program may deadlock on cyclic definitions.

6.5 A Partial Type Order

In this section we define the formal framework for the algorithm that transforms a program with para-functional annotations into a well-typed program in the parallel core language.

The algorithm operates on program functions over the following set of types.

Definition 13 (Type) *T is a type if and only if*

1. *(Atomic) $T = \mathrm{Int}$ or $T = \mathrm{Bool}$ or*
2. *(Lists) $T = [U]$ for some type U or*

$$\text{Task}\ \frac{T \to_{\mathrm{T}} U}{T \to_{\mathrm{T}} \mathrm{Task}(U)} \qquad \text{Tuple}\ \frac{T_1 \to_{\mathrm{T}} U_1, \ldots, T_n \to_{\mathrm{T}} U_n}{(T_1, \ldots, T_n) \to_{\mathrm{T}} (U_1, \ldots, U_n)}$$

$$\text{List}\ \frac{T \to_{\mathrm{T}} U}{[T] \to_{\mathrm{T}} [U]} \qquad \text{Stream}\ \frac{T \to_{\mathrm{T}} U}{[T] \to_{\mathrm{T}} \mathrm{Stream}(U)}$$

Figure 6.2: A Type Relation

3. *(Tuples)* $T = (T_1, \ldots, T_n)$ *for some types* T_i *and* $n > 0$,
4. *(Tasks)* $T = \mathrm{Task}(U)$ *for some type* U,
5. *(Streams)* $T = \mathrm{Stream}(U)$ *for some type* U.

Among types we introduce a partial order as follows:

Definition 14 (Type Order) *The binary type relation* $\to_{\mathrm{T}}$ *is the reflexive transitive closure of the derivation rules given in Figure 6.2:* $T \to_{\mathrm{T}} T'$ *holds if and only if there is a sequence of* $n \geq 0$ *derivations* $T = T_0 \to_{\mathrm{T}} T_1 \to_{\mathrm{T}} \ldots \to_{\mathrm{T}} T_n = T'$.

We use the same symbol $\to_{\mathrm{T}}$ for a single application of the derivation rule as for its reflexive transitive closure; the distinction should be clear from the context.

The intuition for this relation is stated by the following

Lemma 3 (Synchronization/Extension) *If* $T \to_{\mathrm{T}} U$, *there exists an extension function*

$$\mathrm{ext}[T, U] = e : T \to U$$

and a corresponding synchronization function

$$\mathrm{sync}[U, T] = s : U \to T$$

such that

1. e *does not use any* `wait` *operations,*
2. s *does not use any* `start` *operations,*
3. *for any* $v : T$, $s(e(v)) = v$ *and, for any* $v' : U$, $e(s(v')) = v'$.

The formulation of this lemma is somewhat sloppy by mixing the object-level of program functions with the meta-level of their denotations (thus neglecting some technical details). We will ignore this distinction in the rest of this section.

Proof: The proof proceeds by induction over the structure of U. We only consider (as a sample case) $U = \mathrm{Stream}(U')$. Then we have (assuming $U \neq T$) $T = [T']$ for some T' and may define

```
sync[U,T](x) =
  scase x of
    snil => snil
    scons(h,t) => sync[U',T'](h):sync[U,T](wait(t))
```

The other cases proceed in a similar fashion. □

Consequently we may define two operations $\sqcap_{\mathrm{T}}$ and $\sqcup_{\mathrm{T}}$ on types:

Definition 15 (Intersection Type) *The intersection $T_1 \sqcap_{\mathrm{T}} T_2$ of two types T_1 and T_2 is that type T such that*

$$\begin{aligned} &T \rightharpoonup_{\mathrm{T}} T_1, T \rightharpoonup_{\mathrm{T}} T_2, \\ &\forall U.(U \rightharpoonup_{\mathrm{T}} T_1, U \rightharpoonup_{\mathrm{T}} T_2) \to T \not\rightharpoonup_{\mathrm{T}} U \end{aligned}$$

(if such a T exists).

Definition 16 (Union Type) *The union $T_1 \sqcup_{\mathrm{T}} T_2$ of two types T_1 and T_2 is that type T such that*

$$\begin{aligned} &T_1 \rightharpoonup_{\mathrm{T}} T, T_2 \rightharpoonup_{\mathrm{T}} T, \\ &\forall U.(T_1 \rightharpoonup_{\mathrm{T}} U, T_2 \rightharpoonup_{\mathrm{T}} U) \to U \not\rightharpoonup_{\mathrm{T}} T \end{aligned}$$

(if such a T exists).

Lemma 4 (Uniqueness) *If $T_1 \sqcap_{\mathrm{T}} T_2$ (respectively $T_1 \sqcup_{\mathrm{T}} T_2$) exists, it is unique.*

Proof: We proceed by induction over the structure of T_1. Assume T_1 is an atomic type. If $T_1 \rightharpoonup_{\mathrm{T}} T_2$ then $T_1 \sqcap_{\mathrm{T}} T_2 = T_1 \sqcup_{\mathrm{T}} T_2 = T_1$. Otherwise $T_1 \sqcap_{\mathrm{T}} T_2$ and $T_1 \sqcup_{\mathrm{T}} T_2$ do not exist.

Assume T_1 is a compound type and we have two distinct types U_1 and U_2 that fulfill the definition of $T_1 \sqcap_{\mathrm{T}} T_2$. We then have the picture

$$\begin{tikzcd} T_1 & T_2 \\ U_1 \arrow[u] \arrow[ur] & U_2 \arrow[u] \arrow[ul] \end{tikzcd}$$

We may assume that $T_1 \not\rightharpoonup_{\mathrm{T}} T_2$ (and $T_2 \not\rightharpoonup_{\mathrm{T}} T_1$) since otherwise $U_1 = U_2 = T_1$ (respectively $U_1 = U_2 = T_2$).

We may also assume that in $U_1 \rightharpoonup_{\mathrm{T}} \ldots \rightharpoonup_{\mathrm{T}} T' \rightharpoonup_{\mathrm{T}} T_1$ and $U_2 \rightharpoonup_{\mathrm{T}} \ldots \rightharpoonup_{\mathrm{T}} S' \rightharpoonup_{\mathrm{T}} T_1$ all types in the first derivation sequence are distinct from all types in the second one (in particular $T' \neq S'$). Otherwise let T_1 be the first common type in both sequences. We now show a contradiction.

1. If $T_1 = \mathrm{Task}(T)$ we have a contradiction $T' = S' = T$.
2. If $T_1 = \mathrm{Stream}(T)$ we have a contradiction $T' = S' = [T]$.
3. If $T_1 = [a]$, then $U_1 = [b]$ with $b \rightharpoonup_{\mathrm{T}} a$ and $U_2 = [c]$ with $c \rightharpoonup_{\mathrm{T}} a$. By induction we may construct a unique $V = [b \sqcup_{\mathrm{T}} c]$. By the definition of $\rightharpoonup_{\mathrm{T}}$ we know that $T_2 = \mathrm{Task}^j([d])$ or $T_2 = \mathrm{Task}^j(\mathrm{Stream}(d))$ for some d and $j \geq 0$. By definition of $\sqcup_{\mathrm{T}}$ we thus have $V \rightharpoonup_{\mathrm{T}} T_2$ but also $V \rightharpoonup_{\mathrm{T}} T_1$. By assumption (no common types in the derivation sequence) we know that $T_1 \rightharpoonup_{\mathrm{T}} V$ and consequently $T_1 = V$. This yields the contradiction $T_1 \rightharpoonup_{\mathrm{T}} T_2$.
4. If $T_1 = (a_1, \ldots, a_n)$ then we have $U_1 = (b_1, \ldots, b_n)$ and $U_2 = (c_1, \ldots, c_n)$ with $b_i \rightharpoonup_{\mathrm{T}} a_i$ and $c_i \rightharpoonup_{\mathrm{T}} a_i$ for all i. By induction we may construct a unique $V = (b_1 \sqcup_{\mathrm{T}} c_1, \ldots, b_n \sqcup_{\mathrm{T}} c_n)$. By definition of $\rightharpoonup_{\mathrm{T}}$ we know that $T_2 = \mathrm{Task}^j((d_1, \ldots, d_n))$ for some d_i and $j \geq 0$. By definition of $\sqcup_{\mathrm{T}}$ we have $V \rightharpoonup_{\mathrm{T}} T_2$ but also $V \rightharpoonup_{\mathrm{T}} T_1$. By assumption (no common types in the derivation sequence) we know that $T_1 \rightharpoonup_{\mathrm{T}} V$ and consequently $T_1 = V$. This yields the contradiction $T_1 \rightharpoonup_{\mathrm{T}} T_2$.

The induction step for $T_1 \sqcup_{\mathrm{T}} T_2$ proceeds analogously. $\square$

Under this partial order, para-functional type annotations only increase the types of expressions. Correspondingly the typing algorithm proceeds in several iterations, each of which may only increase the types of expressions in the program.

To guarantee termination of the algorithm, it is necessary to introduce the following

Definition 17 (Maximum Extension Type) *The maximum extension* $\max_{\mathrm{T}}(T, T')$ *of a type* T *in direction* T' *(with* $T \rightarrow_{\mathrm{T}} T'$*) is the largest type* U *with* $T \rightarrow_{\mathrm{T}} U \rightarrow_{\mathrm{T}} T'$ *such that* U *does not contain any subtype* $\mathrm{Task}(\mathrm{Task}(V))$*. If already* T *contains such a subtype, then we define* $U := T$*.*

The basic motivation for this definitions is that it makes hardly sense to have a task that returns just another task as its result (the task might as well wait for the result of the other task without loss of potential parallelism).

Lemma 5 (Uniqueness) *For every pair of types* $T \rightarrow_{\mathrm{T}} T'$*,* $\max_{\mathrm{T}}(T, T')$ *exists and is unique.*

Proof: We proceed by induction over the structure of T. If T is atomic, then either $T' = T = \max_{\mathrm{T}}(T, T')$ or $T' = \mathrm{Task}^i(T)$ and $\max_{\mathrm{T}}(T, T') = \mathrm{Task}(T)$.

In case $T = [U]$ we have $T' = \mathrm{Task}^i(\mathrm{List}(U'))$ (or $T' = \mathrm{Task}^i(\mathrm{Stream}(U'))$) for some $U \rightarrow_{\mathrm{T}} U'$ and $i \geq 0$. Let $V = \max_{\mathrm{T}}(U, U')$.

If V contains a subtype with two tasks, then

$$\max_{\mathrm{T}}(T) = [V] \text{ (respectively } \max_{\mathrm{T}}(T) = \mathrm{Stream}(V))$$

Otherwise

$$\max_{\mathrm{T}}(T) = \mathrm{Task}([V]) \text{ (respectively } \max_{\mathrm{T}}(T) = \mathrm{Task}(\mathrm{Stream}(V)))$$

The other cases proceed in a similar fashion. □

Finally we need the following

Definition 18 (Maximum Extension Set) *The maximum extension* $\max_{\mathrm{T}}(T)$ *of a type* T *is*

$$\{\max_{\mathrm{T}}(T, T') | T \rightarrow_{\mathrm{T}} T'\}$$

The transformation algorithm described in Section 6.6 requires this upper bound for the types of function arguments/results to guarantee termination.

Lemma 6 (Finiteness) *For every* T*,* $\max_{\mathrm{T}}(T)$ *is finite.*

Proof: There is only a finite number of rules for the extension of T. The "extension tree" with root T has therefore finite degree in every node. In every path of this tree there exists some T_m such that $\max_{\mathrm{T}}(T, T') = T_m$ for all T' with $T_m \rightarrow_{\mathrm{T}} T'$. The union of the sets all types $T \rightarrow_{\mathrm{T}} \ldots \rightarrow_{\mathrm{T}} T_m$ in all paths is finite and contains all maximum extension types. □

6.6 The Transformation Algorithm

The transformation algorithm proceeds in a sequence of iterations each of which applies a set of transformation rules to the program. During the first iteration, the para-functional annotations are removed; synchronization and extension functions are inserted to yield a well-typed program in the core language. The goal of the following iterations is to defer synchronizations as far as possible to maximize parallelism:

1. Synchronization of *function arguments* is delegated from the caller of a function to the callee: in the application $f(a)$ of a function $f : S \rightarrow_{\mathrm{T}} T$, the argument $a : S$ may be transformed into some $a'' : S''$ (with $S \rightarrow_{\mathrm{T}} S''$). In this case, the application is replaced by $f'(sync[S'', S'](a''))$ where

$$f' : S' \rightarrow T; f'(x) = \ f(sync[S', S](x))$$

where $S' := \max_{\mathrm{T}}(S, S'')$ i.e. $S \rightarrow_{\mathrm{T}} S' \rightarrow_{\mathrm{T}} S''$. The definition of f' will be transformed to a more efficient one where the synchronization of the argument x is deferred.

2. Synchronization of *function results* is delegated from the callee to the caller: in a function definition

$$f : S \rightarrow T$$
$$f(x) = e$$

the body $e : T$ may be transformed into some $e'' : T'$ (with $T \rightarrow_{\mathrm{T}} T''$). In this case, the transformation algorithm replaces the definition by

$$f : S \rightarrow T; f(x) = \ \mathit{sync}[T', T](f'(x))$$
$$f' : S \rightarrow T'; f'(x) = \ \mathit{sync}[T'', T'](e'')$$

where $T' := \max_{\mathrm{T}}(T, T'')$ i.e. $T \rightarrow_{\mathrm{T}} T' \rightarrow_{\mathrm{T}} T''$. Further applications of f are replaced by applications of f' avoiding the additional synchronization.

This process proceeds until a "fixpoint" is reached and the program does not change any more. In more detail, the algorithm uses the following sets of functions:

1. **Extension.** $\mathit{ext}[T, T']$ (with $T \rightarrow_{\mathrm{T}} T'$) is the extension function $e : T \rightarrow T'$.

2. **Synchronization.** $\mathit{sync}[T', T]$ (with $T \rightarrow_{\mathrm{T}} T'$) is the synchronization function $s : T' \rightarrow T$.

3. **Synchronized Constants.** $\mathit{syncval}[T', T, c']$ (with $T \rightarrow_{\mathrm{T}} T'$) is a constant c originally defined as $c = e$ with $e : T$. However, e has been transformed into e'' with $e'' : T''$ and the definition of c has been replaced by

$$\mathit{syncret}[T', T, c'] := c$$
$$c = \mathit{sync}[T', T](c')$$
$$c' = \mathit{sync}[T'', T'](e'')$$

where $T'' = \max_{\mathrm{T}}(T, T')$ i.e. $T \rightarrow_{\mathrm{T}} T' \rightarrow_{\mathrm{T}} T''$. Every further occurrence of c is replaced by an occurrence of c'.

4. **Synchronized Results.** $\mathit{syncret}[T', T, f']$ (with $T \rightarrow_{\mathrm{T}} T'$) is a function $f : S \rightarrow T$ originally defined as $f(x) = e$ with $e : T$. During the transformation, however, e has been extended into $e' : T'$ and the definition of f has been replaced by

$$\mathit{syncret}[T', T, f'] := f$$
$$f(x) = \mathit{sync}[T', T](f'(x))$$
$$f'(x) = \mathit{sync}[T'', T'](e'')$$

where $T'' = \max_{\mathrm{T}}(T, T')$ i.e. $T \rightarrow_{\mathrm{T}} T' \rightarrow_{\mathrm{T}} T''$. Every further application of f is replaced by an application of f'.

5. **Synchronized Arguments.** $\mathit{syncarg}[T', T, f]$ (with $T \rightarrow_{\mathrm{T}} T'$) is a function f with some application $f(a)$ whose argument type has been extended from T to T'.

$$\mathit{syncarg}[T', T, f'] := f$$
$$f'(x) = f(\mathrm{sync}[T', T](x)$$

The definition of f' is later replaced by a more efficient one where argument synchronization does not necessarily take place before f is called.

We now give the skeleton of the transformation algorithm.

Algorithm 1 (Transformation Algorithm)

1. **Transformation** Apply the transformation rules defined in Section 6.7.

   By the rule for function applications, applications of $f = syncret[T', T, f']$ are replaced by f'. Likewise applications $f(a)$ with $f : S \to T$, $a : S'$ and $S \to S'$ are replaced by $f'(a)$ with $f = syncarg[S', S, f']$.

   During this step, new functions $sync[T', T]$ (synchronization functions), $ext[T, T']$ (extension functions), $syncval[T', T, c']$ (synchronized constant values), $syncret[T', T, f']$ (synchronized return values) and $syncarg[T', T, f']$ (synchronized argument values) may be generated.

2. **Simplification** Simplify all applications $ext[T, T''](sync[T', T](a))$ with $T \to_{\mathrm{T}} T'$ and $T \to_{\mathrm{T}} T''$ (but not necessarily $T' \to_{\mathrm{T}} T''$)

   If $T' = T''$, then the result is a. If $T' \to_{\mathrm{T}} T''$, then the result is $ext[T', T''](a)$. If $T'' \to_{\mathrm{T}} T'$, then the result is $sync[T', T''](a)$.

   Otherwise, find the largest T''' with $T \to_{\mathrm{T}} T'''$, $T''' \to T'$ and $T''' \to T''$. The simplified result then is $ext[T''', T''](sync[T', T'''](a))$.

   In a similar fashion, simplify all applications $sync[T, T''](ext[T', T](a))$.

   During this step, new functions $ext[T, T']$ and $sync[T', T]$ may be generated.

3. **Argument Extension** Process every function definition $f''(x) = f(sync[T'', T](x))$ with $f = syncarg[T'', T, f'']$ generating a new function function $f' : T' \to S'$ with $T \to_{\mathrm{T}} T' \to_{\mathrm{T}} T''$ and $S \to_{\mathrm{T}} S'$ such that

   $$\begin{aligned} & syncret[S', S, f'''] := f'' \\ & f''(x) = sync[S', S](f'''(x)) \\ & f'''(x) = f'(sync[T'', T'](x))) \end{aligned}$$

   During this step, new $sync[T', T]$, $ext[T, T']$, $syncval[T', T', c']$, $syncret[T', T, f']$ and $syncarg[T', T, f']$ may be generated.

4. **Iteration** Repeat the whole process until no more $syncval[T', T, c']$, $syncret[T', T, f']$ and $syncarg[T', T, f']$ functions are generated.

5. **Result Extensions** Transform all applications $ext[T, T'](f(a))$ (with $f : S \to T$) into $f'(a)$ (with $f' : S \to T'$) such that the extended compound data structure of type T' is constructed *without* computing the intermediate structure of type T.

The application of the transformation rules raises some subtle points concerning termination. Consider e.g. the following program

```
f(1) where
  f :: Int -> Int
  f(x) = f(x)@
```

Without any precautions, the program might be infinitely expanded to

```
f ::  Int -> Task(T1); f(x) = start(f1(x))
f1 ::  Int -> T1=Task(T2); f1(x) = start(f2(x))
f2 ::  Int -> T2=Task(T3); f2(x) = start(f3(x))
...
```

The basic problem is that expanding the annotation `@` into a `start` call in every level of recursion yields (for this program) a function that has no type that can be described with a finite number of `Task` constructions. In this sense above program is not "reasonable".

A compiler must, however, terminate for every annotated program. Our approach to solve this problem is not to extend function results/arguments beyond the maximum extension type. The correctness of above algorithm is then stated by the following

Lemma 7 (Correctness) *Let P be a functional program and P' the corresponding program without para-functional annotations. If the transformation algorithm terminates on input P, it has transformed P into a well-typed program P'' in the core language that produces the same result as P' (provided that P' terminates).*

Lemma 8 (Termination) *The transformation algorithm terminates on every program P.*

Proof: For proving correctness, we show that after every step of the algorithm, the resulting program P'' is well-typed and produces the same result as the "stripped" version P' of P (assuming that P terminates).

1. **Transformation:** By Lemma 9, every translated expression is well-typed and consistent with the original. Furthermore, the types of constants and the interfaces of functions do not change; consequently P'' produces the same result as P'.
2. **Simplification:** By Lemmas 10 and 11, simplification is correct and terminates.
3. **Argument Extension:** By Lemma 12 and 13, the transformation of the expression $syncarg[T', T, f](x)$ does not change the result and terminates.
4. **Result Extension:** By Lemma 14 and 14, the transformation of $ext[T, T'](f(a))$ does not change the result and terminates.

For proving termination, it now suffices to show that from every original program function $f : S \rightarrow T$ only a finite number of *syncarg*/*syncret* functions can be (directly or indirectly) generated:

1. By transformation rules "Function Applications" and "Task Creations" (see Section 6.7) we have $U \in \max_{\mathrm{T}}(S)$ for every $f' : U \rightarrow T$ with $f = syncarg[U, S, f']$.
2. By transformation rule "Function Definitions" (see Section 6.7) we have $U \in \max_{\mathrm{T}}(T)$ for every $f' : S \rightarrow U$ with $f = syncret[U, T, f']$.

By Lemma 6 (Finiteness), $\max_{\mathrm{T}}(S)$ and $\max_{\mathrm{T}}(T)$ are finite and so is the set of functions directly or indirectly generated from f in the algorithm phases "Transformation" and "Argument Extension". □

6.7 The Transformation Rules

In this subsection we define the system of transformation rules to be applied in the "Transformation" phase of the algorithm. The basic invariant of these rules is given by the following

Lemma 9 (Transformation) *The application of a transformation rule to an expression $e : T$ yields an expression $e' : T'$ where*

1. *$T \rightarrow_{\mathrm{T}} T'$ (types may not decrease).*
2. *e' is in the parallel core language and well typed,*

3. $\mathbf{TE}[\![sync[T', T](e')]\!] = \mathbf{TE}[\![e]\!]$ *(the semantics is consistent),*

4. *the types of constants and the interfaces of functions do not change.*

Proof: By construction in the following sections. □

The transformations are given in the form

$$e \vdash d : T$$

which reads as "e is transformed into d of type T".

We assume that all program identifiers have been appropriately renamed such that each occurrence of the same identifier refers to the same definition.

Furthermore, we use in the definitions of these rules the sets *syncval*, *syncret* and *syncarg* as "global variables" i.e. they are not explicitly threaded through the rules but only mentioned if they actually affect the rule. For instance,

$$\frac{\begin{array}{c}\dots\\ c = \mathrm{syncval}[T', T, c']\end{array}}{\dots}$$

states an additional premise where the variables T', T, c' are existentially quantified. It thus reads "if there are some T', T, c' such that ...". Correspondingly

$$\frac{\dots}{\begin{array}{c}\dots\\ c = \mathit{syncval}[T', T, c']\end{array}}$$

extends the environment by an additional assertion between c, T', T and a new identifier c'.

This non-standard treatment of the environment is justified by the fact that the results and interfaces of functions and constants do not change by the transformations. In particular, in a definition block `let` d `in` e the subexpression e might be well processed before d is handled. The iterated application of the transformations by the main algorithm guarantees that any rule that becomes applicable by an extension of the environment is eventually applied.

6.7.1 Task Annotations

A para-annotated expression e (with free variables $x_1, \dots, x_n$) is abstracted into a function $f = \underline{\lambda} x_1 \dots \underline{\lambda} x_n . e$ from which a `start` expression is constructed:

$$\frac{e \vdash d : T}{e\texttt{@} \;\vdash\; (\texttt{let } f(x_1\texttt{, ..., } x_n\texttt{) = } d \texttt{ in start(}f\texttt{, } x_1\texttt{, ..., } x_n\texttt{))} : \mathit{Task}(T)}$$

6.7.2 Stream Annotations

A stream annotation transforms a list into a stream. The stream construction is constrained by that, assuming the stream head is of type T, the stream tail has to be of type Task(Stream(T)).

If the tail expression is a list, there are two cases:

If the head and the tail can be extended to a common supertype V, they are appropriately adjusted. This is the usual case (typically $T_h = U$ i.e. a list is transformed into a stream of the same base type):

$$\frac{e_h \vdash d_h : T_h, e_t \vdash d_t : T_t \quad T_t = \mathrm{Task}^i(S), S = [U], V = T_h \sqcup_{\mathrm{T}} U, W = \mathrm{Stream}(V)}{e_h \texttt{ :@ } e_t \ \vdash\ (\mathrm{ext}[T_h, V](d_h) \texttt{ :s } \mathrm{ext}[T_t, \mathrm{Task}(W)](d_t)) : W}$$

Otherwise both the head and the tail are synchronized to a common subtype V. This is the consequence of an inconsistent user annotation: a list is extended into a stream whose elements are extended to two different task types (with no common supertype). In this case, the compiler should probably issue a warning.

$$\frac{e_h \vdash d_h : T_h, e_t \vdash d_t : T_t \quad T_t = \mathrm{Task}^i(S), S = [U], V = T_h \sqcap_{\mathrm{T}} U, W = \mathrm{Stream}(U)}{e_h \texttt{ :@ } e_t \ \vdash\ (\mathrm{ext}[V, U](\mathrm{sync}[T_h, V](d_h)) \texttt{ :s } \mathrm{ext}[S, \mathrm{Task}(W)](\mathrm{sync}[T_t, S](d_t))) : W}$$

Finally the tail may have been already extended to some stream type. In this case, the head type T_h should match the base type U of the tail (such that we do not have to synchronize and traverse the structure of the tail stream). If not, the head is adjusted (which generally requires synchronization to a common subtype V). If $V \neq T_h$ and $V \neq U$, the compiler should probably issue a warning.

$$\frac{e_h \vdash d_h : T_h, e_t \vdash d_t : T_t \quad T_t = \mathrm{Task}^i(S), S = \mathrm{Stream}(U), V = T_h \sqcap_{\mathrm{T}} U}{e_h \texttt{ :@ } e_t \ \vdash\ (\mathrm{ext}(V, U)(\mathrm{sync}[T_h, V](d_h)) \texttt{ :s } \mathrm{ext}[S, \mathrm{Task}(S)](\texttt{sync}[T_t, S](d_t))) : S}$$

6.7.3 Synchronization Annotations

The base expression is synchronized from T to its ground type U which does neither contain 'Task' nor 'Stream' constructors (streams are synchronized to lists).

$$\frac{e \vdash d : T \quad U = \mathrm{ground}(T)}{e\texttt{!} \ \vdash\ \mathrm{sync}[T, U](d) : U}$$

6.7.4 Type Annotations

Constants whose types are annotated as T' are extended from T to T'. If not $T \rightarrow_{\mathrm{T}} T'$, the annotation is ignored (the compiler then should probably issue a warning, probably also if T' is larger than the maximum extension of T). See also the explanations in the next paragraph.

$$\frac{e \vdash d : T, a \vdash b : B \quad T \rightarrow_{\mathrm{T}} T', U = \max_{\mathrm{T}}(S, T')}{\begin{array}{c}\texttt{let para } c \texttt{ :: } T'\texttt{; } c^S \texttt{ = } e \texttt{ in } a \ \vdash \\ (\texttt{let } c'^U \texttt{ = } sync[T', U](ext[T, T'](d))\texttt{,} c^S \texttt{ = } \mathrm{sync}[U, S](c') \texttt{ in } b) : B \\ syncval\,[U, S, c'] = c\end{array}}$$

6.7.5 Constant Definitions

If the value of a constant c is extended from type S to type T, a new identifier c' receives the value extended to $U = \max_{\mathrm{T}}(S, T)$. c is defined by synchronization of c'; the association is remembered in *syncval*. Further applications of c will be replaced by applications of c'.

$$\frac{e \vdash d : T, a \vdash b : B \qquad U = \max_{\mathrm{T}}(S,T)}{\texttt{let } c^S \texttt{ = } e \texttt{ in } a \ \vdash\ (\texttt{let } c'^U \texttt{ = } sync[T,U](d)\texttt{,} c^S \texttt{ = } \mathrm{sync}[U,S](c') \texttt{ in } b) : B}$$
$$syncval[U,S,c'] = c$$

6.7.6 Identifiers

If for some identifier an extended "buddy" is defined, this buddy replaces the synchronized value. Consequently synchronization is deferred to a later moment (and possibly shifted from the current task into another tasks). In a later optimization phase, repeated synchronizations of the same identifier are combined (see Subsection 6.11).

$$\frac{c = syncval[U,T,c']}{c \vdash\ c' : U}$$

6.7.7 Function Definitions

If the body of a function f is extended from S_i to T_i, a new function f' receives the value extended to $U_i = \max_{\mathrm{T}}(S_i, T_i)$. f is defined by synchronization of f'; the association is remembered in *syncret*. Further applications of f will be replaced by applications of f'.

$$\frac{e \vdash d : T, a \vdash b : B \qquad U = \max_{\mathrm{T}}(S,T)}{\begin{array}{c}\texttt{let } f^{R\to S}(x_j) \texttt{ = } e \texttt{ in } a \ \vdash \\ (\texttt{let } f'^{R\to U}(x) \texttt{ = } sync[T,U](d)\texttt{, } f^{R\to S}(x) \texttt{ = } \mathrm{sync}[U,S](f'(x)) \texttt{ in } b) : B\end{array}}$$
$$syncret[U,S,f'] = f$$

6.7.8 Function Applications

We give two rules for transforming function applications, where the first one (if applicable) is applied before the second one.

If a function f is defined by some buddy f' with an extended result, the application of f is replaced:

$$\frac{f = syncret[T,S,f']}{f\texttt{(}e\texttt{)} \ \vdash\ f'\texttt{(}e\texttt{)} : T}$$

If the types of function arguments are extended beyond the types of the function parameters, a new function is created that accepts the extended arguments without synchronization (up to the maximum extension type).

$$\frac{e \vdash d : T \qquad U = \max_{\mathrm{T}}(S,T)}{f^{S\to R}\texttt{(}e\texttt{)} \ \vdash\ \texttt{let } f'^{U\to R}\texttt{(}x\texttt{) = } f\texttt{(}sync[U,S](x)\texttt{) in } f'\texttt{(}sync[T,U](d)\texttt{)} : R}$$
$$syncarg[U,S,f'] = f$$

6.7.9 Conditionals

The two branches of a conditional have to be of the same type. If the branch expressions have a common supertype T, they are appropriately extended:

$$\frac{\begin{array}{c} e_c \vdash d_c : T_c, e_t \vdash d_t : T_t, e_f \vdash d_f : T_f \\ T = T_t \sqcup_{\mathrm{T}} T_f \end{array}}{\begin{array}{c} \texttt{if } e_c \texttt{ then } e_t \texttt{ else } e_f \ \vdash \\ (\texttt{if } \mathrm{sync}[T_c, \mathrm{Bool}](d_c) \texttt{ then } \mathrm{ext}[T_t, T](d_t) \texttt{ else } \mathrm{ext}[T_t, T](d_f)) : T \end{array}}$$

Otherwise both branches are synchronized to a common subtye. In our implementation of the typing algorithm, only this rule is applied after the first iteration to restrict parallelism in iterated compositions with multiple iterators.

$$\frac{\begin{array}{c} e_c \vdash d_c : T_c, e_t \vdash d_t : T_t, e_f \vdash d_f : T_f \\ T = T_t \sqcap_{\mathrm{T}} T_f \end{array}}{\begin{array}{c} \texttt{if } e_c \texttt{ then } e_t \texttt{ else } e_f \ \vdash \\ (\texttt{if } \mathrm{sync}[T_c, \mathrm{Bool}](d_c) \texttt{ then } \mathrm{sync}[T_t, T](d_t) \texttt{ else } \mathrm{sync}[T_t, T](d_f)) : T \end{array}}$$

6.7.10 List/Stream Selection

Both the list selector `case` and the stream selector `scase` are compiled into the lower level constructs `isnil`, `head`, `tail`. These operators need their arguments to be synchronized to a list (respectively stream) cell.

$$\frac{\begin{array}{c} e \vdash d : T \\ T = \mathrm{Task}^i(S), S = [U] \vee S = \mathrm{Stream}(U) \end{array}}{\texttt{isnil}(e) \ \vdash \ \texttt{isnil}(\mathrm{sync}[T, S](d)) : \mathrm{Bool}}$$

$$\frac{\begin{array}{c} e \vdash d : T \\ T = \mathrm{Task}^i(S), S = [U] \vee S = \mathrm{Stream}(U) \end{array}}{\texttt{head}(e) \ \vdash \ \texttt{head}(\mathrm{sync}[T, S](d)) : U}$$

$$\frac{\begin{array}{c} e \vdash d : T \\ T = \mathrm{Task}^i(S), S = [U] \end{array}}{\texttt{tail}(e) \ \vdash \ \texttt{tail}(\mathrm{sync}[T, S](d)) : S}$$

$$\frac{\begin{array}{c} e \vdash d : T \\ T = \mathrm{Task}^i(S), S = \mathrm{Stream}(U) \end{array}}{\texttt{tail}(e) \ \vdash \ \texttt{tail}(\mathrm{sync}[T, S](d)) : \mathrm{Task}(S)}$$

6.7.11 List Construction

The head and the tail of a list construction have to be of the same base type. If a common supertype exists, both head and tail are extended:

$$\frac{\begin{array}{c} e_h \vdash d_h : T_h, e_t \vdash d_t : T_t \\ T_t = \mathrm{Task}^i(S), S = [U], V = T_h \sqcup_{\mathrm{T}} U \end{array}}{e_h \ : \ e_t \ \vdash \ (\mathrm{ext}[T_h, V](d_h) \ : \ \mathrm{ext}[T_t, [V]](d_t)) : [V]}$$

Otherwise, the head and the tail are synchronized to the maximum common subtype:

$$\frac{\begin{array}{c} e_h \vdash d_h : T_h, e_t \vdash d_t : T_t \\ T_t = \mathrm{Task}^i(S), S = [U], V = T_h \sqcap_{\mathrm{T}} U \end{array}}{e_h \ : \ e_t \ \vdash \ (\mathrm{sync}[T_h, V](d_h) \ : \ \mathrm{sync}[T_t, [V]](d_t)) : [V]}$$

If the tail is a stream, we adapt the head to the base type U of the tail (after a possible synchronization to the intersection type V). If $V \neq T_h$ and $V \neq U$, the compiler should probably issue a warning.

$$\frac{\begin{array}{c} e_h \vdash d_h : T_h, e_t \vdash d_t : T_t \\ T_t = \mathrm{Task}^i(S), S = \mathrm{Stream}(U), V = T_h \sqcap_{\mathrm{T}} U \end{array}}{e_h \ \texttt{:} \ e_t \ \vdash \ (\mathrm{ext}(V,U)(\mathrm{sync}[T_h,V](d_h)) \ \texttt{:}_{\texttt{S}} \ \mathrm{ext}[S,\mathrm{Task}(S)](\mathrm{sync}[T_t,S](d_t))) : S}$$

6.7.12 Stream Construction

In the case of a stream construction, the tail must be already a stream type. Since we do not want to (synchronize and) traverse the structure of this tail, the head type is adapted to the base type U of the tail (possibly via the intersection type V). If $V \neq T_h$ and $V \neq U$, the compiler should probably issue a warning.

$$\frac{\begin{array}{c} e_h \vdash d_h : T_h, e_t \vdash d_t : T_t \\ T_t = \mathrm{Task}^i(S), S = \mathrm{Stream}(U), V = T_h \sqcap_{\mathrm{T}} U \end{array}}{e_h \ \texttt{:}_{\texttt{S}} \ e_t \ \vdash \ (\mathrm{ext}(V,U)(\mathrm{sync}[T_h,V](d_h)) \ \texttt{:}_{\texttt{S}} \ \mathrm{ext}[S,\mathrm{Task}(S)](\mathrm{sync}[T_t,S](d_t))) : S}$$

6.7.13 Tuple Selection

The basis expression must be synchronized to a tuple type.

$$\frac{\begin{array}{c} e \vdash d : T \\ T = \mathrm{Task}^i(S), S = (S_1, \ldots, S_i, \ldots, S_n) \end{array}}{e\,\texttt{.}\,i \ \vdash \ (\mathrm{sync}[T,S](d)\,\texttt{.}\,i) : S_i}$$

6.7.14 Tuple Construction

The types of the tuple components are not constrained at all:

$$\frac{\begin{array}{c} e_1 \vdash d_1 : T_1, \ldots, e_n \vdash d_n : T_n \\ T = (T_1, \ldots, T_n) \end{array}}{(e_1 \texttt{,} \ \ldots \texttt{,} \ e_n) \ \vdash \ (d_1 \texttt{,} \ \ldots \texttt{,} \ d_n) : T}$$

6.7.15 Unary and Binary Expressions

The arguments of strict (arithmetic and logical) operators are synchronized to the appropriate ground type. Two examples are given below:

$$\frac{e_1 \vdash d_1 : T_1, e_n \vdash d_n : T_n}{e_1 \ \texttt{+} \ e_2 \ \vdash \ (\mathrm{sync}[T_1,\mathrm{Int}](d_1) \ \texttt{+} \ \mathrm{sync}[T_2,\mathrm{Int}](d_2)) : \mathrm{Int}}$$

$$\frac{e \vdash d : T}{\texttt{\^{}} \ e \ \vdash \ (\texttt{\^{}} \ \mathrm{sync}[T,\mathrm{Bool}](d)) : \mathrm{Bool}}$$

6.7.16 Constants

$$\frac{}{\mathit{intconst} \ \vdash \ \mathit{intconst} : \mathrm{Int}} \qquad \frac{}{\mathit{boolconst} \ \vdash \ \mathit{boolconst} : \mathrm{Bool}}$$

$$\frac{}{\texttt{[ ]} \ \vdash \ \texttt{[ ]} : \mathrm{Nil}} \qquad \frac{}{\texttt{[ ]}_{\texttt{S}} \ \vdash \ \texttt{[ ]}_{\texttt{S}} : \mathrm{SNil}}$$

6.7.17 Task Creation

`start` expressions are treated similar to function applications;

$$\frac{f = \mathit{syncret}[T, S, f']}{\texttt{start(}f\texttt{, } e\texttt{)} \vdash \texttt{start(}f'\texttt{, } e\texttt{)} : \mathrm{Task}(T)}$$

Extended task arguments are never synchronized before the task is started; instead an auxiliary function takes the free variables of the arguments and performs the synchronization within the function.

$$\frac{\begin{array}{c} e \vdash d : T \\ U = \max_{\mathrm{T}}(S, T) \end{array}}{\begin{array}{ll} \texttt{start(}fS \to R\texttt{, } e\texttt{)} \vdash & \texttt{let } f'^{U \to R}\texttt{(}x\texttt{) = } f\texttt{(}\mathit{sync}[U, S](x)\texttt{);} \\ & \quad g^{X_j \to R}\texttt{(}x_j\texttt{) = } f'\texttt{(}\mathit{sync}[T, U](e)\texttt{)} \\ & \texttt{in start(}g\texttt{, } x_j\texttt{)} : R \\ & \mathit{syncarg}[U, S, f'] = f \end{array}}$$

6.7.18 Task Synchronization

By a `wait` expression one 'Task' level is removed from an expression type:

$$\frac{\begin{array}{c} e \vdash d : T \\ T = \mathrm{Task}(S) \end{array}}{\texttt{wait(}e\texttt{)} \vdash \texttt{wait(}d\texttt{)} : S}$$

6.7.19 Non-Deterministic Synchronization

The task list argument of `any` has to be appropriately synchronized:

$$\frac{\begin{array}{c} e \vdash d : T \\ T = \mathrm{Task}^i(\mathrm{List}(S)) \lor T = \mathrm{Task}^i(\mathrm{Stream}(S)) \end{array}}{\texttt{any(}e\texttt{)} \vdash \texttt{any(}\mathrm{sync}[T, \mathrm{List}(S)](d)\texttt{)} : (S, \mathrm{List}(S))}$$

6.7.20 Task Abortion

The task list argument of `stop` (but not its result) has to be appropriately synchronized:

$$\frac{\begin{array}{c} e_t \vdash d_t : T_t, e_r \vdash d_r : T_r \\ T_t = \mathrm{Task}^i(\mathrm{List}(S)) \lor T_t = \mathrm{Task}^i(\mathrm{Stream}(S)) \end{array}}{\texttt{stop } e_t \texttt{ in } e_r \vdash (\texttt{stop } \mathrm{sync}[T_t, \mathrm{List}(S)](d_t) \texttt{ in } d_r) : T_r}$$

6.8 Simplifications

In the "Simplification" phase of the algorithm, nested applications of *sync* and/or *ext* applications are reduced as denoted by the following rules:

$$\frac{\mathit{ext}[T, T](e)}{e} \qquad \frac{\mathit{sync}[T, T](e)}{e}$$

$$\frac{ext[T',T''](ext[T,T'](e))}{ext[T,T''](e)} \qquad \frac{sync[T',T](sync[T'',T'](e))}{sync[T'',T](e)}$$

$$\frac{\begin{array}{c} ext[T,T'](sync[T'',T](e)) \\ T'' \rightarrow_{\mathrm{T}} T' \end{array}}{ext[T'',T'](e)} \qquad \frac{\begin{array}{c} ext[T',T](sync[T'',T'](e)) \\ T \rightarrow_{\mathrm{T}} T'' \end{array}}{sync[T'',T](e)}$$

$$\frac{\begin{array}{c} sync[T',T''](ext[T,T'](e)) \\ T \rightarrow_{\mathrm{T}} T'' \end{array}}{ext[T,T''](e)} \qquad \frac{\begin{array}{c} sync[T',T](ext[T'',T'](e)) \\ T \rightarrow_{\mathrm{T}} T'' \end{array}}{sync[T'',T](e)}$$

Lemma 10 (Correctness) *The application of the simplification rules to an expression does not change the result.*

Lemma 11 (Termination) *The application of the simplification rules terminates.*

Proof: Correctness is a direct consequence Lemma 3. Since the application of a rule removes one level of *sync*/*ext* nestings, the algorithm terminates. □

6.9 Argument Extension

During the "Transformation" step, a set of functions $syncarg[T'',T,f']$ is generated where

$$f'(x) = f(\mathrm{sync}[T'',T](x))$$

The goal in the "Argument Extension" step is to transform f' into a counterpart of f where the synchronization of the arguments is not performed before the function is called but deferred to those expressions within f where synchronization is actually required. However, a naive "inlining" of a_j would lead to inefficient programs since the same task might be synchronized multiple times. Therefore it is better to use some intermediate strategy where synchronization is partially done before and partially inside the function activation yielding a definition

$$\begin{aligned} f'(x) &= f''(\mathrm{sync}[T'',T'](x)) \\ f''(x) &= \ldots \end{aligned}$$

where T' is the maximum argument type which it "makes sense" to pass to f without synchronization.

We construct f'' by creating a copy of f where the parameter types are extended from T to U. We apply the set of transformation rules to the body of f in the same fashion as described in the first step of the transformation algorithm. Thus a well-typed definition for f'' is generated where synchronization is performed at the basic operations or by introducing additional *syncarg* functions.

Lemma 12 (Correctness) *Argument extension of a function f does neither change the type interface nor the result of f.*

Proof: By Lemma 9. □

The computation of T' (and thus the construction of a synchronization schedule) is based on the following intuition:

1. If in f the synchronization of some task becomes *only* necessary in the base case of recursion, then it is better to pass the unsynchronized task to the function (and consequently forward it through all recursion levels). Thus the execution of the task may overlap with the recursion.
2. However, if the synchronization of some task becomes necessary in some recursive branches, then it is better to synchronize the task *before* the function is called to avoid repeated (and consequently unnecessary) synchronizations of the same task.

We implement this intuition by the following algorithm that takes $f : T \to S$ and a type T'' and returns the maximum T' with $T \to_{\mathrm{T}} T' \to_{\mathrm{T}} T''$ which it "makes sense" to pass to f without synchronization.

Algorithm 2 (Maximum Argument Type)

1. **Extension Space:** Construct the type space

   $$\{T' | T \to_{\mathrm{T}} T' \to_{\mathrm{T}} \max_{\mathrm{T}}(T, T'')\}$$

   and sort it in an topological order with respect to $\to_{\mathrm{T}}$.
2. **Test:** Let U be the first element in the type space. Replace the outermost [ ] ('List') and the outermost 'Stream' constructors in U by $\overline{\text{List}}$ and $\overline{\text{Stream}}$.

   Push the tuple (f, U) on a stack and simulate the application of the transformation rules to f as described in the first step of the transformation algorithm. During this simulation we check whether additional applications of *sync* (i.e. in further consequence applications of `wait`) are required:

   (a) **Recursive Branches:** Ignore the `wait` calls derived from additional *sync* applications in those branches of f that are not (directly or indirectly) recursive.

   (b) **List Elements:** Ignore the `wait` calls derived from additional *sync* applications when executed on arguments of type $\overline{\text{Task}}$.

   (c) **Head:** If `head` is applied to an argument of type $\overline{\text{List}}(T)$ or $\overline{\text{Stream}}(T)$, the result is of type $\overline{T}$ derived from T by replacing all 'Task' constructors by $\overline{\text{Task}}$.

   (d) **Tail:** If `tail` is applied to an argument of type $\overline{\text{List}}(T)$ respectively $\overline{\text{Stream}}(T)$, the result is of type $\overline{\text{List}}(T)$ respectively $\overline{\text{Stream}}(T)$.

   (e) **Function Application:** For every applications of a function g on an argument type S, check whether the tuple (g, S) is on the stack. If not, check recursively whether g may be called with S' where S' is constructed from S by replacing all $\overline{\text{List}}$ constructors by 'List' (and $\overline{\text{Stream}}$ by 'Stream').
3. **Success:** If no `wait` is introduced by additional *sync* applications, the test succeeds; in this case return T.
4. **Failure:** Otherwise, remove all U' with $U \to_{\mathrm{T}} U'$ from the type space and continue with the next type.

The algorithm is similar to an abstract interpretation of the first step of the transformation algorithm where only the type information is propagated but additional synchronizations are only checked (but not actually introduced).

Lists/streams are treated specially by marking the task types that are retrieved by applications of `head`; their synchronization is allowed within the current function activation. However, the element sequence as a whole must not be synchronized, therefore the $\overline{\text{List}}$ respectively $\overline{\text{Stream}}$ marks are removed when passed to another function.

When applying the algorithm, the compiler has already inlined non-recursive functions and even shortcut simple cycles of (mutually recursive) functions. The algorithm is a heuristical one; in practice, it seems to capture the intuition sketched above pretty well. Anyway, the programmer may use the annotation *exp*`!` to explicitly demand synchronization of values before the function is called.

Lemma 13 (Termination) *The algorithm "Maximum Argument Type" terminates.*

Proof: By Lemma 6 (Finiteness) the type space is finite, therefore the outer loop terminates. Furthermore, for every function $f : S \to T$, the set $\max_{\mathrm{T}}(S)$ is finite and so is the depth of the stack used by the inner loop. □

6.10 Result Extension

The goal of "Result Extension" is to remove those applications of $ext[T,T'](e)$ (with $T \to_{\mathrm{T}} T'$) that have not been cancelled in the "Simplification" phase of the algorithm. Specifically we would not like to have a compound data structure a be first constructed and afterwards extended by decomposing e, extending its individual component, and composing the new structure e'. Instead e' should be constructed first-place applying the necessary `start` calls. We now describe the algorithm by defining a set of transformation rules.

First of all, the simplification rules described in Subsection 6.8 are applied before any of the following rules is applied.

If $T' = T$, no change is required at all.

$$ext[T,T](e) \;\vdash\; e$$

If $T' = \mathrm{Task}(T)$, e is abstracted and a `start` call is created:

$$\frac{ext[T,U](e) \vdash d \qquad T' = \mathrm{Task}(U)}{ext[T,T'](e) \;\vdash\; \texttt{let } f(x) \texttt{ = } d \texttt{ in start(} f\texttt{,}x\texttt{)}}$$

Otherwise, the outermost constructor in T' is that of an compound data type (tuple, list, stream). In this case, we proceed by case distinction over the structure of the expression.

In list constructions, the type extension distributes over the head and the tail:

$$\frac{T = [U], T' = [U']}{ext[T,T'](\texttt{[ ]}) \;\vdash\; \texttt{[ ]}}$$

$$\frac{ext[U,U'](e_h) \;\vdash d_h, \;\; ext[T,T'](e_t) \;\vdash d_t \qquad T = [U], T' = [U']}{ext[T,T'](e_h \,\texttt{:}\, e_t) \;\vdash\; d_h \,\texttt{:}\, d_t}$$

This is true for stream constructions as well:

$$\frac{T = [U], T' = \mathrm{Stream}(U')}{ext[T,T'](\texttt{[ ]}) \;\vdash\; \texttt{[ ]}_{\texttt{S}}}$$

$$\frac{ext[U,U'](e_h) \;\vdash\; d_h, \;\; ext[T,\mathrm{Task}(T')](e_t) \;\vdash\; d_t \qquad T = [U], T' = \mathrm{Stream}(U')}{ext[T,T'](e_h \,\texttt{:}\, e_t) \;\vdash\; d_h \,\texttt{:}_{\texttt{S}}\, d_t}$$

An extended list component (head element respectively tail list) requires the extension of the whole list:

$$\frac{ext[[T],[T']](e) \;\vdash\; d}{ext[T,T'](\texttt{head(}e\texttt{)}) \;\vdash\; \texttt{head(}d\texttt{)}}$$

$$\frac{ext[T,T'](e) \;\vdash\; d}{ext[T,T'](\texttt{tail(}e\texttt{)}) \;\vdash \texttt{tail(}d\texttt{)}}$$

In tuple constructions respectively selections, type extensions distribute over the tuple components:

$$\frac{\begin{array}{c} ext[U_i,U'_i](e_i) \;\vdash\; d_i \\ T=(U_i), T'=(U'_i) \end{array}}{ext[T,T'](\texttt{(}e_i\texttt{)}) \;\vdash \texttt{(}d_i\texttt{)}}$$

$$\frac{\begin{array}{c} ext[(U_1,\ldots,T,\ldots,U_n),(U_1,\ldots,T',\ldots,T_n)](e) \;\vdash\; d \\ \mathrm{type}(e)=(U_1,\ldots,T,\ldots,U_n) \end{array}}{ext[T,T'](e\texttt{.}i) \;\vdash\; d\texttt{.}i}$$

In conditionals, both branches have to be extended. In definition blocks, only the block expression is affected.

$$\frac{ext[T,T'](e_t) \;\vdash\; d_t,\;\; ext[T,T'](e_f) \;\vdash\; d_f}{ext[T,T'](\texttt{if } e_c \texttt{ then } e_t \texttt{ else } e_f) \;\vdash\; \texttt{if } e_c \texttt{ then } d_t \texttt{ else } d_f}$$

$$\frac{ext[T,T'](e) \;\vdash\; d}{ext[T,T'](\texttt{let } defs \texttt{ in } e) \;\vdash\; \texttt{let } defs \texttt{ in } d}$$

If a local constant has to be extended, the original value must be preserved as well since it might be used in another place. If the original value (which is now defined as a synchronization of the extended value) is not any more used, it is removed in a later phase of the compilation.

$$\frac{ext[T,T'](e) \;\vdash\; d}{\begin{array}{c}\texttt{let } c \texttt{ = } e \texttt{ in } \ldots ext[T,T'](c) \ldots \;\vdash \\ \texttt{let } c \texttt{ = } sync[T',T](c')\texttt{; } c' = d \texttt{ in } \ldots\; c'\ldots\end{array}}$$

If an extended function result is required, a new function is created whose body is extended:

$$\frac{ext[T,T'](e) \;\vdash\; d}{\begin{array}{c}\texttt{let } f\texttt{(}x_i\texttt{) = } e \texttt{ in } \ldots ext[T,T'](f\texttt{(}e_i\texttt{)}) \ldots \;\vdash \\ \texttt{let } f\texttt{(}x_i\texttt{) = } e\texttt{; } f'\texttt{(}x_i\texttt{) = } d \texttt{ in } \ldots f'\texttt{(}e_i\texttt{)} \ldots\end{array}}$$

Task creations need not be investigated separately, since the extension type is always a task, which has been covered by one of the initial rules.

In task terminations, only the result expression is affected by the type extension.

$$\frac{ext[T,T'](e_r) \;\vdash\; d_r}{ext[T,T'](\texttt{stop } e_t \texttt{ in } e_r) \;\vdash\; \texttt{stop } e_t \texttt{ in } d_r}$$

The application of these rules basically proceeds from more complicated expressions to simpler ones until the level of local constants, parameters, and function applications is reached. Parameters of a function cannot be extended any more (unless we would extend all applications of the function). However, local constants are extended by extending their definitions. Likewise, extending function applications yield new functions that have to be (recursively) extended as well. Termination is guaranteed, however, by the following

Algorithm 3 (Result Extension) Process an expression $ext[T,T'](e)$ (with $T \rightarrow_{\mathrm{T}} T'$) as follows:

1. Apply above transformation rules from top to bottom.
2. If the application of a function f has to be extended to from result type T to T', check whether a tuple (f, f', T') is on the stack. In this case, replace f by f'. Otherwise create a function f' as a copy of f, push (f, f', T') on the stack and extend the body of f' recursively from T to T'.
3. Proceed in the same fashion for extending the definitions of local constants.

Lemma 14 (Correctness) *The result extension algorithm applied to an expression e does not change the result of e.*

Lemma 15 (Termination) *The result extension algorithm terminates.*

Proof: The correctness of the algorithm is a direct consequence of the transformation rules. For proving termination, it suffices to show that the rules for (possibly recursive) function and constant definitions do not lead to an infinite sequence of type extensions.

The number of functions and of function applications in the program is finite and no identical tuples are on the stack. Consequently, if the algorithm runs into non-termination, an infinite number of tuples $(f_0, f_1, T_1), (f_1, f_2, T_2), \ldots$ must be on the stack with $T_i \rightarrow_{\mathrm{T}} T_{i+1}$.

The only extension rule, however, that increases types is the rule for stream construction where the extension of a list cell to $T = \mathrm{Stream}(U)$ requires the extension of the list tail to $\mathrm{Task}(T)$. By the rule for the extension of task types, however, this is immediately reduced to an extension to T. Therefore types cannot increase, which yields a contradiction. $\Box$

6.11 Local Optimizations

In the parallel core language, the evaluation of textually independent subexpressions is subject to the following constraints only:

1. **Conditionals** The selection expression is evaluated before the corresponding branch expression is chosen and evaluated.
2. **Local Definitions** The sequence of definitions is evaluated in the stated order before the body expression is evaluated.

In order to maximize parallelism, we apply to the body of every function a number of optimizations concerning the order of task creations and synchronizations.

6.11.1 Common Synchronization Expressions

The transformation algorithm defers synchronization operations as much as possible; typically they only occur at the argument positions of strict basis operations. However, then the same task may be synchronized multiple times within a function body. We eliminate multiple occurrences of common synchronization expressions as follows:

$$\mathbf{CS}\left[\!\!\left[\begin{array}{l}\ldots\\ \boxed{sync(e)\ \ldots\ sync(e)}\\ \ldots\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\ldots\\ \texttt{let x = }sync(\mathrm{e})\\ \texttt{in }\boxed{x\ \ldots\ x}\\ \ldots\end{array}\right]\!\!\right]$$

The framed box here denotes the smallest expression that contains all occurrences of the common synchronization operation (not considering occurrences in different conditional branches). For instance, the expression

```
if wait(x)
  then wait(x)*wait(x)+wait(y)
  else wait(y)
```

is optimized to

```
let t = wait(x)
in if t
  then t*t+wait(y)
  else wait(y)
```

6.11.2 Late Synchronizations

While the transformation algorithm defers synchronization operations as much as possible, it does actually *not* guarantee that a synchronization occurs at the latest possible moment. E.g. in an expression

$$f(\texttt{wait(}e_1\texttt{)},e_2\texttt{)}$$

the synchronization of e_1 should be performed *after* e_2 has been evaluated. Practically, this is a problem if e_2 is the application of a recursive function (the only expression whose evaluation does not take a small amount of time). Consequently, function applications are drawn out of expressions that contain synchronization operations:

$$\mathbf{LS}\left[\!\!\left[\begin{array}{l}\ldots\\ \boxed{sync(e_1)\ \ldots\ f(e_2)}\\ \ldots\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\ldots\\ \texttt{let}\ x\ \texttt{=}\ f(e_2)\\ \texttt{in}\ \boxed{sync(e_1)\ \ldots\ x}\\ \ldots\end{array}\right]\!\!\right]$$

The frame denotes the largest expression that does neither contain a conditional nor a block expression surrounding both $sync(e_1)$ and e_2. If this expression is a block expression, the new constant definition is added to the statement sequence.

For instance, the expression

```
f(wait(a) + g(wait(b),c) +h(d))
```

is optimized to

```
let x = g(wait(b),c)
    y = h(d)
in f(wait(a)+x+y)
```

The optimization of the order of constant definitions is described later in this section.

As a special case we handle recursive applications of the form

$$\begin{array}{l} f\texttt{(...) =}\\ \quad\ldots\\ \quad\boxed{e_2\ \texttt{:}\ f\texttt{(}e_1\texttt{)}}\end{array}$$

where the framed expression denotes the result of the function.

The code generator transform such recursive function applications into a tail-recursion from (and consequently into a loop) where e_1 is executed before the next iteration (the tail-recursive call of f) is started. We prefer the benefit of loop iteration over the parallelism between the recursion and the evaluation of e_1. Consequently, above rule is not applied to this special case.

6.11.3 Early Task Creations

The counterpart of the optimization described in the previous subsection is the transformation of expressions that contain `start` expressions. For instance, in the expression

e_1 `+ start(`f`,` e_2`)`

the `start` call should occur *before* e_1 is evaluated. The corresponding transformation is

$$\mathbf{ET}\left[\!\!\left[\begin{array}{l} \dots \\ \boxed{e_1 \ \dots\ \texttt{start}(f,\ e_2)} \\ \dots \end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l} \dots \\ \texttt{let}\ x = \texttt{start}(f,\ e_2) \\ \texttt{in}\ \boxed{e_1 \dots x} \\ \dots \end{array}\right]\!\!\right]$$

The frame denotes the largest expression that does neither contain a conditional nor a block expression surrounding both e_1 and the `start` call. If this expression is a block expression, the new constant definition is added to the statement sequence.

6.11.4 Order of Constant Definitions

The previously described transformations yield block expressions that consist of task definitions mixed with ordinary definitions that however contain synchronization operations. In order to maximize parallelism, the `start` calls should occur in such a sequence as early as possible and the expressions with `wait` operations as late as possible.

However, due to data dependencies (e.g. a `start` call may depend on a constant whose definition contains a `wait`), both goals may contradict each other. To overcome this problem, we choose a simple heuristical approach that performs well in typical cases.

We define the order of operations by a topological sort (with respect to data dependencies) such that the next definition is selected (if multiple choices exists) by applying the following criteria in the denoted order:

1. Select a task creation statement.
2. Select a constant definition without synchronization operations.
3. Select that definition on which most remaining definitions depend.

Since the basic semantics of the (para-)functional language is strict, the correctness of the program is actually not affected by the selection. However, in the case of recursive constant definitions, it is essential that tasks are started *before* synchronization are performed.

For instance, the expression

```
let x = g(wait(b), c)
    y = h(d)
    z = start(f, a)
in f(wait(a)+x+y*wait(z))
```

is optimized to

```
let z = start(f, a)
    y = h(d)
    x = g(wait(b),c)
in f(wait(a)+x+y*wait(z))
```

A final optimization is

$$\mathbf{OC}\left[\!\!\left[\begin{array}{l}\ldots\\ t\ \texttt{=}\ \texttt{start(}f\texttt{,}\ a_i\texttt{)}\\ \boxed{\texttt{wait(}t\texttt{)}}\\ \ldots\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\ldots\\ \boxed{f\texttt{(}a_i\texttt{)}}\\ \ldots\end{array}\right]\!\!\right]$$

assuming that t is not in use anywhere else.

Chapter 7

Parallel Compositions

In this chapter, we present parallel variants of the iterated compositions introduced in Section 4.2. These parallel forms generalize the notion of "data parallelism" by integrating both phases of value generation and value combination without intermediate synchronization and construction of temporary data structures. The dynamical behavior of the individual parallel variants is analyzed in detail in Section 7.7.

Part of this chapter is taken from [Schreiner, 1994b].

7.1 Overview

Figure 7.1 lists several variants of parallel composition forms whose translation is described in this chapter. The programmer indicates parallelism by annotations `|@`par_i where the parallelization condition par_i evaluates to true if the corresponding iteration level is to be parallelized. The token `|` is interpreted as `|@(False)`; `|@` denotes `|@(True)`.

In general, par_i may depend on free variables, even on variables that are defined within the composition and whose value changes from one iteration to the next. In the current implementation, however, such general parallelization conditions are only supported for tree compositions and non-deterministic compositions.

The interpretation of a parallelization annotation depends on the form of the composition (see Figure 7.2 and the translations in the following sections):

exp	$\rightarrow$	*composer iterator*	Composition
composer	$\rightarrow$	`comp` (f, b)	Standard Composition
	$\rightarrow$	`rcomp` (f, b)	Reverse Composition
	|	`tcomp` (f , b, p)	Tree Composition
	|	`ndcomp` (f , b, p)	Non-Deterministic Compos.
iterator	$\rightarrow$	`[` *exp* `|` $iter_1$ `|@`par_1 ... `|@`par_n $iter_n$ `]`	Iterator List

Figure 7.1: Parallel Compositions

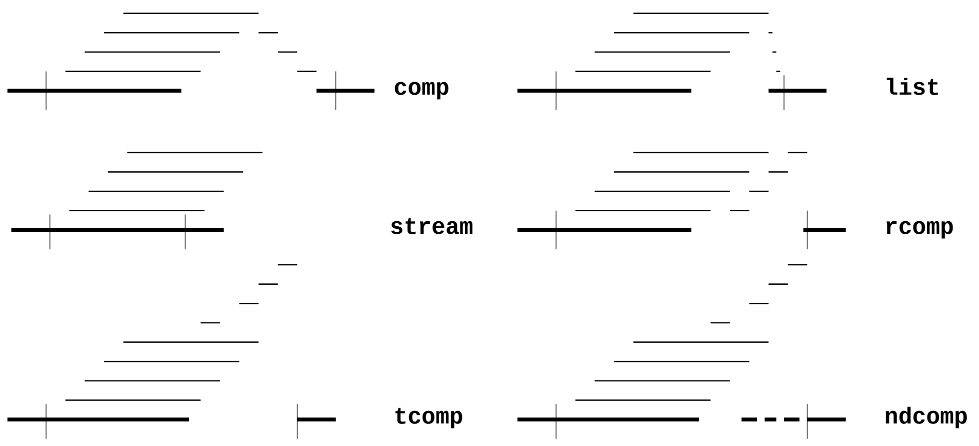


Figure 7.2: Parallel Composition Forms

1. **Standard Compositions:** In a `comp` form, the current task starts a successor task to recursively generate and process the subsequent bindings of the variables in $iter_i$. The current task processes with the current bindings of $iter_i$ all remaining iterators $iter_{i+1}$ to $iter_n$ and so constructs a sequence of values.

   It then waits for the termination of the successor task and combines its result with the values it has generated. Consequently, the individual bindings for the variables $iter_i$ are processed in parallel but the combinations are performed sequentially.

   There are two special forms of standard compositions:

   (a) **Lists:** The composer `list` is defined as `comp(:,[ ])`. However, due to the special processing of the : operator (see Section 4.4.3), each composition is performed immediately when a new value of the sequence is generated and *before* the tail list is constructed.

   The task finally waits for the result list of the child task which is written into the tail of the last cell. Consequently, the individual list segments are truly generated in parallel and there is only a constant-time operation required to perform the final composition.

   (b) **Streams:** The composer `stream` is defined as `comp(:`$_S$`,[ ]`$_S$`)`. Consequently, the current task does not wait for the result of the child task but writes the task handle into the tail of the cell and returns immediately.

   Since streams are homogeneous data structures (the tail of every stream cell contains a task), basically every iterator requires an annotation `|@`. However, the programmer may skip the annotations because the transformation algorithm described in Section 6.10 performs the necessary task extensions automatically.

2. **Reverse Compositions:** In a `rcomp` form, the main task starts a task for the current variable bindings of $iter_i$ and passes to the new task the handle of the task that has processed the *previous* bindings.

   Consequently, all tasks are started by the main task but it is *not* the main task that waits for the individual results. Instead synchronization is deferred to the individual subtasks where the last one returns the total result.

3. **Tree Compositions:** In a `tcomp` form, the combination function $f : T \times T \to T$ must be an *associative* operation. The `|@`*par* annotation splits the generated value sequence $a_1, \ldots, a_n$ into clusters $A_i = f(a_0, f(\ldots, f(a_{j-1}, a_j)))$ each of which is computed by a separate task.

   The individual clusters are then combined in parallel in the form of a binary tree

   $$f((f(A_0, A_1), f(A_2, A_3)), \ldots, f(f(A_{n-3}, A_{n-2}), f(A_{n-1}, b)))$$

   The (optional) termination predicate p asserts that the result does not change any more and combination may be terminated prematurely.

4. **Non-Deterministic Compositions:** In a `ndcomp` composition, the combination function $f : T \times T \to T$ must be an *associative and commutative* operation. The `|@`*par* annotation splits the generated value sequence $a_1, \ldots, a_n$ into clusters $A_i = f(a_0, f(\ldots, f(a_{j-1}, a_j)))$ each of which is computed by a separate task.

   The individual clusters are then combined non-deterministically to

   $$f((f(A_{p_0}, A_{p_1}), f(A_{p_2}, A_{p_3})), \ldots, f(f(A_{p_{n-3}}, A_{p_{n-2}}), f(A_{p_{n-1}}, b)))$$

   for some permutation p. The (optional) termination predicate p asserts that the result does not change any more and combination may be terminated prematurely.

In Figure 7.1, the dynamic behavior of the individual composition forms is sketched where each line represents the execution of a task. The vertical bounds indicate how long the main task is involved in the parallel composition i.e. when execution starts and when the result is returned.

In the `stream` case, the result contains handles to tasks that need not have terminated yet (this is in general true for any `comp` operation with a combination function f that is non-strict in its second argument).

The details of the translation are described in the following sections. The dynamical behavior of the constructs is analyzed in detail in Section 7.7.

7.2 Examples

The following program utilizes consumer producer-parallelism between a sequence of tasks creating a stream of values $g(x)$ and a task that sequentially processes these elements by a function g:

```
list [ g(y) | y <- s ] where
  s = stream [ f(x) | x <- 1..n ]@
```

The additional `@` annotation decouples the stream construction totally from the current task (otherwise the first stream element $f(1)$ is computed by the current task).

The compiler generates from this a program in the core language similar to the following

```
consume(wait(t)) where
  t = start(produce, 1, n)
  produce :: (Int,Int) -> Stream(Int)
  produce(i,n) =
    if i > n
      then snil
      else scons(f(x), start(produce, i+1, n))
```

```
consume :: Stream(Int) -> [Int]
consume(s) =
  scase s of
    nil => nil
    y:t  => let z = g(y)
            in z:consume(wait(t))
```

Compared to a corresponding sequential program

```
consume(produce(1,n)) where
  produce :: (Int,Int) -> [Int]
  produce(i,n) =
    if i > n
      then [ ]
      else f(x):produce(i+1, n)
  consume :: [Int] -> [Int]
  consume(s) =
    case s of
      [ ] => [ ]
      y:t  => g(y):consume(t)
```

a variety of modifications are necessary: `start` and `wait` calls are inserted, list constructions are replaced by stream constructions and the interfaces of functions are changed.

The program

```
list [ y | x <- 1..n |@ y <- fact(x) ]
```

constructs a list that contains the factors of all numbers from 1 to n. The outer iteration level is parallelized; the resulting C target code has essentially the following structure:

```
  F(x) {
    if (x > n)
      return(NIL);
    else {
      TASK t = START1(F, x+1);
      return(G(x, fact(x), t));
    }
  }
  G(x, l, t) {
    LIST r;
    while (!ISNIL(l)) {
      r = RCONS(HEAD(l), &r);
      l = TAIL(l);
    }
    *r = WAIT(t);
    return(r);
  }
}
```

In the program

```
tcomp(prod, 1) [ y | x <- l, i <- 1.. |@(i%5==0) y <- fact(x) ]
```

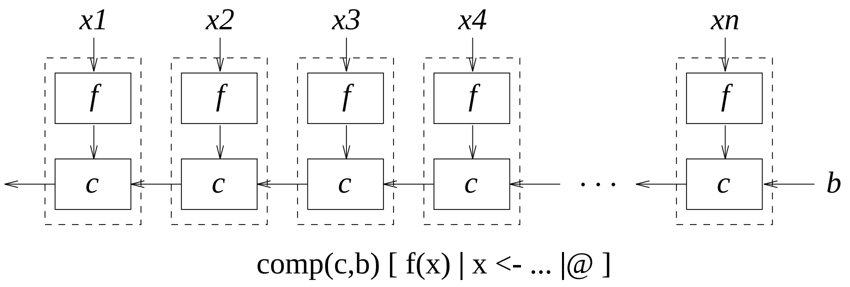


Figure 7.3: Parallel Standard Composition

a list of numbers is factored and all factors are multiplied. For every cluster of five numbers a task is started that computes the corresponding factors and product sequentially. The intermediate results are multiplied in parallel by a binary task tree.

The function

```
forall(l,p) = ndcomp(and, True, not) [ p(x) | x <- l |@ ]
```

tests whether a predicate holds for all elements of a list. Every element is tested by a separate task; if an element is found for which the test fails, the other tasks are aborted and the composition terminates (speculative parallelism).

7.3 Parallel Standard Compositions

In Figure 7.4 we define the general translation of a parallel composition of type `comb(`f`, `b`)`. The derived compositions `list` respectively `stream` are defined as

list = comp(:, [ ])
stream = comp($:_S$, $[\ ]_S$)

Compared to the standard composition (see Section 4.2.3), the translation of the parallel composition requires only small modifications: We simply split the recursive branch by a conditional with test condition *par* and annotate the recursive application of F in that branch in which *par* yields true.

The annotated expression becomes the argument of another function. Therefore the transformation algorithm attempts to generate a new variant of this function where the task argument is not immediately synchronized. Since this function is in general the result of another iterator, the task argument is only synchronized in the base case and task evaluation can proceed in parallel with the recursion. In the last iterator, the amount of parallelism depends on f. In the worst case, only the evaluation of e may proceed in parallel with the child task.

As a simple example with one iterator, the composition form

comp(c,b) [f(x) | x = a; p(x); x' = s(x) |@]

is translated into a function

$$\mathbf{TC}[\![\texttt{comp}(f,b)\ \texttt{[}\ e\ \texttt{|}\ I\ \texttt{]}]\!] = \mathbf{TI}[\![I]\!][\![f]\!][\![e]\!][\![b]\!]$$

$$\mathbf{TI}[\![\ ]\!][\![f]\!][\![e]\!][\![v]\!] \ = \ [\![f\,(e,v)]\!]$$

$$\mathbf{TI}[\![h_i\texttt{:}t_i\texttt{<-}l_i\texttt{,}\ x_j\texttt{<-}a_j\texttt{..}b_j\texttt{+=}c_j\texttt{,}\ y_k\texttt{=}r_k\texttt{,}\ defs_1\texttt{;}\ s\texttt{;}\ y_k\texttt{'}\ \texttt{=}\ u_k\texttt{,}\ defs_2\ \texttt{|@}par\ R]\!]$$
$$[\![f]\!][\![e]\!][\![v]\!] \ =$$

```
[[ let
     F(z_1, ..., z_m, x_1, ..., x_n, y_1, ..., y_o) =
       case z_1 of
         [ ] => v̄
         h_1:t_1 =>
           ...
             case z_m of
               [ ] => v̄
               h_m:t_m =>
                 if x_1 > b̄_1 || ... || x_n > b̄_n
                   then v̄
                   else let
                     defs_1
                   in if ^s
                     then v̄
                     else let
                       defs_2
                     in if par then
                       TI[[R]][[f]][[e]]
                       [[F(t_1, ..., t_m, x_1+c̄_1, ..., x_n+c̄_n, u_1, ..., u_o)@]]
                     else
                       TI[[R]][[f]][[e]]
                       [[F(t_1, ..., t_m, x_1+c̄_1, ..., x_n+c̄_n, u_1, ..., u_o)]]
     b̄_j =b_j; c̄_j = c_j; v̄ = v
   in
     F(l_1, ..., l_m, a_1, ..., a_n, r_1, ..., r_o)                         ]]
```

Figure 7.4: Parallel Composition

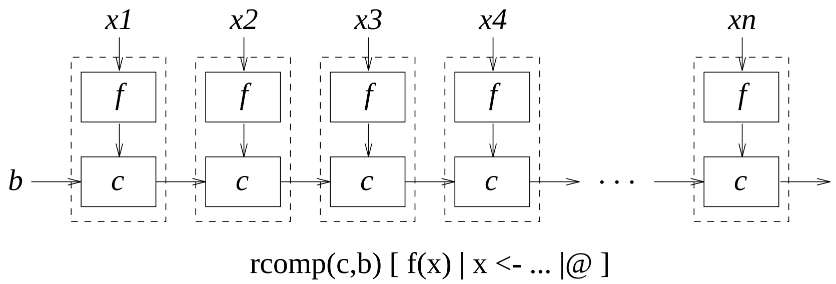


Figure 7.5: Parallel Reverse Composition

```
F(x) =
  if p(x)
    then c(f(x), F(s(x))@)
    else b
```

for which essentially the following C code is generated

```
F(x) {
  if (p(x)) {
    tr = start(F,s(x))
    y = f(x)
    return(c(y, wait(tr)))
  }
  else
    return(b)
}
```

Actually, the new task does not execute F but a function F' with $F'(x) = F(s(x))$ i.e. the successor value is computed within the new task. For multiple iterators, the call of c in the definition of F is replaced by a recursive function G that takes the task argument without synchronization.

7.4 Parallel Reverse Compositions

In Figure 7.6 we define the general translation of a parallel composition of type `rcomb(`f`, `b`)`. Again the core of the body is split into two branches; in contrast to the `comp` case, it does not make sense to annotate the (tail-)recursive function invocation; instead the result of the recursive transformation is determined to execute in parallel with the recursion of F.

For example, the form

`rcomp(`c`,`b`) [ `f`(x) | x = `a`; `p`(x); x' = `s`(x) |@ ]`

is translated into

```
F(x, r) =
  if p(x)
    then F(s(x), c(f(x),r)@)
    else b
```

$$\mathbf{TC}[\![\texttt{rcomp}(f,b)\ \texttt{[}\ e\ \texttt{|}\ I\ \texttt{]}]\!]\ \texttt{=}\ \mathbf{TJ}[\![I]\!][\![f]\!][\![e]\!][\![b]\!]$$

$$\mathbf{TJ}[\![\ ]\!][\![f]\!][\![e]\!][\![v]\!]\ =\ [\![f\texttt{(}e\texttt{,}v\texttt{)}]\!]$$

$$\mathbf{TJ}[\![h_i\texttt{:}t_i\texttt{<-}l_i\texttt{,}\ x_j\texttt{<-}a_j\texttt{..}b_j\texttt{+=}c_j\texttt{,}\ y_k\texttt{=}u_k\texttt{,}\ defs_1\texttt{;}\ s\texttt{;}\ y_k\texttt{'}\ \texttt{=}\ r_k\texttt{,}\ defs_2\ \texttt{|@}par\ R]\!]$$
$$[\![f]\!][\![e]\!][\![v]\!]\ =$$

$$
\left[\!\!\left[
\begin{array}{l}
\texttt{let} \\
\quad F\texttt{(}z_1\texttt{,}\ \ldots\texttt{,}\ z_m\texttt{,}\ x_1\texttt{,}\ \ldots\texttt{,}\ x_n\texttt{,}\ y_1\texttt{,}\ \ldots\texttt{,}\ y_o\texttt{,}\ w\texttt{)}\ \texttt{=} \\
\quad\quad \texttt{case}\ z_1\ \texttt{of} \\
\quad\quad\quad \texttt{[ ] =>}\ w \\
\quad\quad\quad h_1\texttt{:}t_1\ \texttt{=>} \\
\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad \texttt{case}\ z_m\ \texttt{of} \\
\quad\quad\quad\quad\quad\quad \texttt{[ ] =>}\ w \\
\quad\quad\quad\quad\quad\quad h_m\texttt{:}t_m\ \texttt{=>} \\
\quad\quad\quad\quad\quad\quad\quad \texttt{if}\ x_1\ \texttt{>}\ \bar{b}_1\ \texttt{||}\ \ldots\ \texttt{||}\ x_n\ \texttt{>}\ \bar{b}_n \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{then}\ \bar{v} \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{else let} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad defs_1 \\
\quad\quad\quad\quad\quad\quad\quad\quad \texttt{in if \^{}}s \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \texttt{then}\ \bar{v} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \texttt{else let} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad defs_2 \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \texttt{in if}\ par\ \texttt{then} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad F\texttt{(}t_1\texttt{,}\ \ldots\texttt{,}\ t_m\texttt{,}\ x_1\texttt{+}\bar{c}_1\texttt{,}\ \ldots\texttt{,}\ x_n\texttt{+}\bar{c}_n\texttt{,}\ r_1\texttt{,}\ \ldots\texttt{,}\ r_o\texttt{,} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \texttt{(}\mathbf{TJ}[\![R]\!][\![f]\!][\![e]\!][\![v]\!]\texttt{)@)} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \texttt{else} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad F\texttt{(}t_1\texttt{,}\ \ldots\texttt{,}\ t_m\texttt{,}\ x_1\texttt{+}\bar{c}_1\texttt{,}\ \ldots\texttt{,}\ x_n\texttt{+}\bar{c}_n\texttt{,}\ r_1\texttt{,}\ \ldots\texttt{,}\ r_o\texttt{,} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{TJ}[\![R]\!][\![f]\!][\![e]\!][\![v]\!]\texttt{)} \\
\quad \bar{b}_j\ \texttt{=}b_j\texttt{;}\ \bar{c}_j\ \texttt{=}\ c_j \\
\texttt{in} \\
\quad F\texttt{(}l_1\texttt{,}\ \ldots\texttt{,}\ l_m\texttt{,}\ a_1\texttt{,}\ \ldots\texttt{,}\ a_n\texttt{,}\ u_1\texttt{,}\ \ldots\texttt{,}\ u_o\texttt{,}\ v\texttt{)}
\end{array}
\right]\!\!\right]
$$

Figure 7.6: Reverse Composition

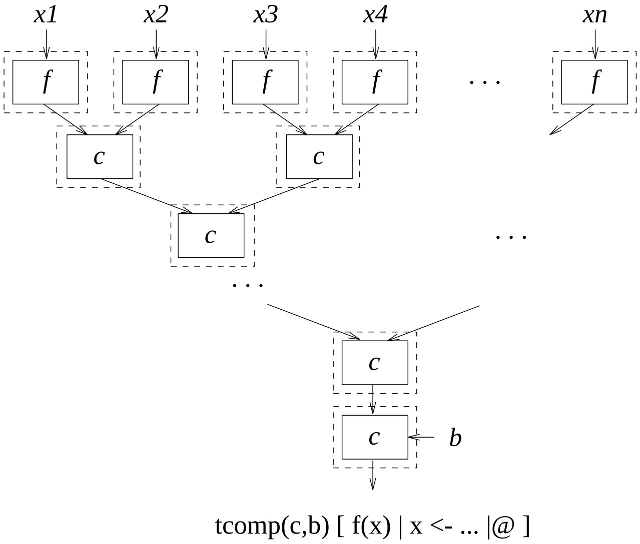


Figure 7.7: Tree Composition

for which essentially the following target code is generated:

```
F(x,rt) {
  while (p(x)) {
    rt = start(C,x,rt)
    x = s(x)
  }
  return(wait(rt))
}

C(x,rt) {
  y = f(x)
  return(c(y,wait(x)))
}
```

7.5 Tree Compositions

7.5.1 Definition

In the composition form

$$\texttt{tcomp(}f\texttt{, }b\texttt{, }p\texttt{) [}e\texttt{ | }I_1\texttt{ |@}par_1\ \ldots\ I_n\texttt{|@}par_n\texttt{]}$$

the expression e is evaluated in every variable binding generated by the iterators $I_1 \ldots I_n$. This yields a sequence of values $a_1 \ldots a_m$ that are combined by the associative combination function $f : T \times T \to T$. Defining $x \circ y := f(x, y)$, the result is

$$a_1 \circ a_2 \circ \ldots \circ a_m \circ b$$

$$\textbf{TC}\ [\![\ \texttt{tcomp}(f,b,p)\ \texttt{[}\ e\ \texttt{|}\ I\ \texttt{]}\ ]\!] = \left[\!\!\left[\begin{array}{l}\texttt{let}\\ \quad \texttt{v = } b\\ \quad \texttt{r = } \textbf{TN}[\![I]\!][\![f]\!][\![e]\!][\![\texttt{v}]\!][\![p]\!]\\ \texttt{in if r.0}\\ \quad \texttt{then } f\texttt{(}r\texttt{.1, v)}\\ \quad \texttt{else v}\end{array}\right]\!\!\right]$$

Figure 7.8: Tree Composition (Top-Level)

where any grouping of the subexpressions by parentheses is allowed and yields the same result. The optional predicate p has the property

$$p(x) \rightarrow (f(x,y) = x \wedge f(y,x) = x)$$

for all x, y. This predicate does not influence the result of the combination but just asserts that the result will not change any more by further combination steps.

The basic idea for implementing this construct is to split the sequence of variable bindings V generated by the iterators I into subsequences $V_1, V_2, \ldots, V_n$. For every V_i a task is started that sequentially evaluates e in all bindings of that sequence and combines the resulting values to $A_i = a_1 \circ a_2 \ldots \circ a_n$.

The main task creates a new task W_i for every pair V_{2i-1}, V_{2i} to perform the combinations that correspond to the leaves of the binary tree. This is done without waiting for the results of the V_i i.e. the synchronizations are deferred to the individual W_i. Then for every pair W_{2j-1}, W_{2j} a task is started to combine the subresults in the next level of the tree; this process continues until one task is left. The main task waits for the result of this task and combines it with b to get the final result. If the predicate p is given, every task tests its arguments for p and skips the combination, if the test succeeds.

The boolean expression `|@`*par* is used by the programmer to control the task creation rate. Whenever *par* yields true, a new binding sequence V_i is started and a new task is created.

7.5.2 The Generation Phase

Figure 7.9 shows the core of the translation:

1. Each iterator I is transformed into a tail-recursive function F (and consequently into a loop by the code generator). Correspondingly the iteration variables are transformed into parameters of F. In detail, these parameters are
   (a) the current bindings of the iteration variables,
   (b) the bindings accumulated since the last task has been created,
   (c) the number of accumulated bindings,
   (d) the list of created tasks.
2. The accumulation of variable bindings is represented as follows:
   (a) For every iteration variable `z` defined by an arithmetic sequence `z <-` a`..`b`+=`c, the lower bound of the iteration subrange is remembered as `zs`.

$$\mathbf{TN}[\![\ ]\!][\![f]\!][\![e]\!][\![v]\!][\![p]\!] \;=\; [\![\ \texttt{(True, } e\texttt{)}\ ]\!]$$

$$\mathbf{TN}[\![h_i\texttt{:}t_i\texttt{<-}l_i\texttt{, } x_j\texttt{<-}a_j\texttt{..}b_j\texttt{+=}c_j\texttt{, } y_k\texttt{=}u_k\texttt{, } defs_1\texttt{; } s\texttt{; } y_k\texttt{' = } r_k\texttt{, } defs_2\ \texttt{|@}par\ R]\!]$$
$$[\![f]\!][\![e]\!][\![v]\!][\![p]\!] \;=$$

```
let
  auxiliary definitions
  G(h_i, t_i, x_j, y_k) =
    let defs_1; defs_2
    in TN[[R]][[f]][[e]][[v]][[p]]
  comb(zs_i, xs_j, ys_k, n) =
    loop( (False, v ) ) [ sum |
      h_i:t_i <- zs_i, x_j <- xs_j .. += c_j, y_k <- rev(zs_k), i <- 1..n,
      sum = (False, v), new = G(h_i, t_i, x_j, y_k),
      next = if sum.0 then
        if new.0 then (True, f(sum.1, new.1)) else sum
        else new;
      ^sum.0 | ^p(sum.1);
      sum' = next ]
  F(z_i, x_j, y_k, zs_i, xs_j, ys_k, n, tasks) =
    case z_i of
      [ ] => red(rev(start(comb, zs_i, xs_j, ys_k, n):tasks))
      h_i:t_i =>
        if x_j > bv_j
          then red(rev(start(comb, zs_i, xs_j, ys_k, n):tasks))
          else let
            defs_1
          in if ^s
            then red(rev(start(comb, zs_i, xs_j, ys_k, n):tasks))
            else let
              defs_2
            in if par
              then F(t_i, x_j+cv_j, r_k, z_i, x_j, [y_k], 1,
                start(comb, zs_i, xs_j, ys_k, n):tasks)
              else F(t_i, x_j+cv_j, r_k, zs_i, xs_j, y_k:ys_k, n+1, tasks)
  lv_j = l_j; av_j = a_j; bv_j = b_j; cv_j = c_j
in
  F(lv_i, av_j, u_k, lv_i, av_j, [ ], 0, [ ])
```

Figure 7.9: Tree Composition (Core)

```
rev(l) = rlist [ x | x <- l ]

red(l) =
  case l of
    [ ] => (False, v)
    h1:t1 =>
      case t1 of
        [ ] => wait(h1)
        h2:t2 => red(start(C, h1, h2) : red0(t2))

red0(l) =
  case l of
    [ ] => [ ]
    h1:t1 => case t1 of
      [ ] => [h1]
      h2:t2 => start(C, h1, h2) : red0(t2)

C(xt, yt) =
  let (xf, xv) = wait(xt)
    in if xf
      then if p(xv)
        then (True, xv)
        else let (yf, yv) = wait(yt)
          in if yf
            then if p(yv)
              then (True, yv)
              else (True, f(xv, yv))
            else (xf, xv)
      else wait(yt)
```

Figure 7.10: Tree Composition (Auxiliaries)

(b) For every iteration variable **x** defined by an list iterator `x <-` l the start of the corresponding sublist is remembered as **xs**.

(c) For every iteration variable **y** defined by a general iteration `y =` a`; ...; y' =` b the sequence of bindings is rememberd (in the reverse order) in a list **ys**.

For every iteration variable therefore two parameters are generated (the current binding and the information about the accumulated bindings). The start values for the current and the accumulated bindings are passed as the initial arguments to F.

3. The transformation of the binding generators is basically the same as for iterated compositions (list iterations are transformed into `case` expressions, arithmetic and general iterations into `if` constructs).

4. If the `|@`*par* expression yields true in the current environment, a new task is created to evaluate the base expression e in all accumulated environments and combine the result values by f; the current environment becomes the first of a new accumulation.

5. Otherwise, the current environment is added to the current accumulation and F starts the next iteration. In the base case of recursion, a last task is started; the results of all tasks are combined by an application of the `red` function.

6. Every task executes a function `comb` that reproduces each environment from the accumulated information, processes the remaining iterators I in this environment, evaluates the base expression e and combines the results.

There are several subtle details in the definition of `comb` that are worth being elaborated:

1. `comb` is implemented by a `loop` composition that it is transformed into a tail-recursive function and finally compiled into a loop (see Section 4.4.2). We cannot use the more elegant `rcomp` form since we need access to the intermediate result for the application of p and for premature termination.

2. Since the main task has already traversed the iteration range and the number of iterations n is known, it is neither necessary to include the upper bounds of arithmetic sequences nor the test condition s. Also the `case` into which list generators are translated (see Section 4.4) is actually not required and might be replaced by low-level applications of `head` and `tail` only (this is not yet done).

3. The definition sequences $defs_1$ and $defs_2$ have to be included into `G` since the remaining iterators I are evaluated in their scope. This cannot be avoided by defining `G` within the scope of the corresponding definitions in F, because `comb` that calls `G` is called also outside of this scope (in the base cases of recursion). All names defined there but not used within I or in e will be later removed by the constant folding phase.

4. The result of a combination may be empty (if the generated sequence of variable bindings is empty). Therefore the result is defined as a tuple whose first element is a flag that denotes whether the second element is a valid result or not (in the second case the variable `v` defined at the top-level is inserted to satisfy the type checker). This is actually a place where an *algebraic (discriminated union) type* would be required (which is not yet supported by the language).

7.5.3 The Combination Phase

Figure 7.10 shows the auxiliary functions that are required in above translation but whose structure is independent of the concrete composition:

1. `rev` reverses its input list; this is required in the function `comb` since the accumulated environments generated by general iterations are stored in the reverse order (by pushing them on top of a list).

2. `red` is called with the list of tasks generated by the composition construct; however this list must be reversed before (to yield the original order of values).

3. `red` and `red2` traverse the task list in a sequence of combination phases. During each phase, pairs of tasks are passed to a new task C that synchronizes and combines their return values. Consequently, the main task creates without synchronization a tree of tasks; the last remaining task in the list denotes the root of the tree whose return value is the result of the combination.

The code generator will transform the recursive functions functions into plain loops that can be efficiently executed. A reverse tree composition variant `trcomp` can be easily implemented by reversing the composition order in the core translation.

7.5.4 Optimizations

The denoted translation scheme yields a program that traverses the iteration range twice; once by the main task and once by the subtasks. In general this is necessary since the variable bindings are generated in sequence. However, in special cases several optimizations are possible (but not yet implemented).

1. If only arithmetic generators are present, then F need not always traverse the whole iteration range. In particular if there is no condition s and the `|@`par condition is of the special form i`%`e`==`c for some constant c and some iteration variable i defined as i `<-` a`..`b, then F may skip $e-1$ iterations by adding e (instead of 1) to the corresponding argument in the recursive call. Consequently the complexity of the sequential path is reduced and in every iteration a task may be started.
2. If only general generators and possibly arithmetic iterators (but not any list generators) are present, then the generated binding list need not be reversed in the `comb` function. Instead the bounds of the arithmetic ranges may be inverted at compile time (transforming `x <- a..b += c` into `x <- b..a -= c`).
3. Additionally the binding list generated from general iterators need not be reversed, if on every iteration of F also the bindings generated by list generators are stored in the reverse fashion (thus effectively reversing the sublists). This would save the extra reverse step but cause an additional list being generated.

Another variation lets even `red` run concurrently with F; a separate task executing `red` may communicate with the main task by a stream transmitting those subtasks that F is generating. Consequently the reduction phase might start before the last of the subtasks has been created and yield more parallelism. However, it is not clear whether the increased communication/synchronization overhead pays off in practice.

7.6 Non-Deterministic Compositions

7.6.1 Motivation

Section 5.5.7 describes the construct `any` that differs from all other core expressions in that its result is not uniquely determined by its input arguments. Instead a class of values is defined of which each member represents an acceptable result. The language semantics thus becomes considerably more complicated.

However there are contexts in which it the final result of an expression is uniquely determined even if the evaluation order is not. Take a commutative and associative operation $\circ$ and a bag of values $A = \{a_1, \ldots, a_n\}$. Then for any permutation $p : \mathbb{N}^n \rightarrow \mathbb{N}^n$, the composition

$$(a_{p_1} \circ (a_{p_2} \circ (\ldots \circ a_{p_n}) \ldots))$$

yields the same result. Consequently any deterministic program implementing this composition is actually *overspecified* by committing to one particular evaluation order.

Especially if the a_i are evaluated by concurrent tasks it might be more efficient to perform the necessary synchronizations in a non-deterministic fashion and combine pairs of elements $a_i \circ a_j$ in the order in which they become available. By making the order of evaluation steps less strict, we will define a construct that may be more efficient than any corresponding construct that is implemented with deterministic operations only.

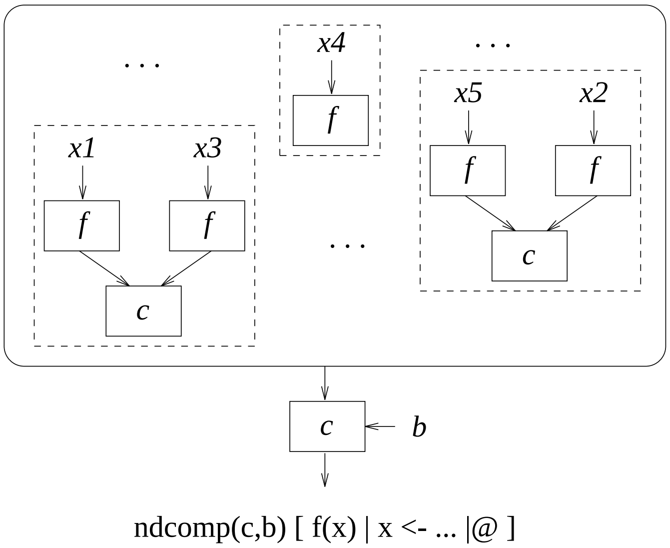


Figure 7.11: Parallel Reverse Composition

7.6.2 Translation

In the composition form

`ndcomp(`f`, `b`, `p`) [ `e` | `I_1` |@`par_1` ... `I_n`|@`par_n` ]`

the expression e is evaluated in every variable binding generated by the iterators $I_1 \dots I_n$, which yields a sequence of values $a_1 \dots a_m$ that are combined by the combination function $f : T \times T \to T$.

The result of this construct is, for some permutation p, the value

$$f(a_{p_1}, f(a_{p_2}, \dots f(a_{p_m}, b) \dots))$$

i.e. the combination may be performed in any order. The optional predicate p asserts (like described in the previous section) that the result will not change any more by further combination steps.

The core translation is the same as for tree compositions except for the combination phase: Here the main task non-deterministically waits for pairs of terminated tasks A_i, A_j and creates a new task to compute $f(A_i, A_j)$; this task is put back into the bag of tasks. All synchronizations are therefore performed by the main task itself. The return value of the last task is combined with the base value b yielding the final result.

Figure 7.10 shows the auxiliary functions that implement the combination phase: `red` and `red2` are mutually recursive functions that perform the non-deterministic composition. They cooperate by removing terminated tasks from the bags and generating new tasks combining each pair of values. If the termination predicate p becomes true, combination is terminated and all remaining tasks are aborted.

```
red(bag) =
  case bag of
    [ ] => (False, v)
    h:t =>
      let ((flag, val), bag2) = any(bag)
        in if flag
          then if p(val)
            then stop bag2 in (True, val)
            else red2(val, bag2)
          else red(bag2)

red2(val, bag) =
  case bag of
    [ ] => (True, val)
    h:t =>
      let ((flag, val2), bag2) = any(bag)
        in if flag
          then if p(val2)
            then stop bag2 in (True, val2)
            else red(start(f, val, val2):bag2)
          else red2(val, bag2)
```

Figure 7.12: Non-Deterministic Composition (Auxiliaries)

7.6.3 Extensions

Currently, `ndcomp` is more restrictive than actually required by that the arguments of f are constrained to the same type. Referring to the basic definition of the result value

$$f(a_{p_1}, f(a_{p_2}, \ldots f(a_{p_m}, b) \ldots))$$

it would make perfect sense to drop this restriction allowing the more general interface $f : S \times T \rightarrow T$. However, in this case the individual reduction operations could not be performed any more in the form of a binary tree. Instead, `red` must be defined e.g. as

```
red(bag) = red2(bag, (False, v))
red2(bag,(flag,val)) =
  case bag of
    [ ] => (False, v)
    h:t => let ((flag2,val2),bag2) = any(bag)
           in if flag
                then if flag2
                       then red2(bag2, (True, f(val, val2)))
                       else red2(bag2, (flag, val))
                else red(bag2, (flag2, val2))
```

i.e. the order of combinations is non-deterministic but the individual combination steps are performed sequentially. It is not clear whether this feature is really required since a mathematical operation $f : S \times T \rightarrow T$ is usually defined by an embedding $g : S \rightarrow T$ and an operator $f' : T \times T \rightarrow T$ as

$$f(x, y) = f'(g(x), y)$$

Therefore a non-deterministic composition

`ndcomp(`f`,` b`) [` e `|` Is`]`

may be easily redefined as

`ndcomp(`f'`,` b`) [` $g(e)$ `|` Is`]`

by the programmer.

7.6.4 Referential Transparency

The `ndcomp` construct does in general not return a unique value; still it is deterministic in certain contexts as stated by the following lemma.

Lemma 16 (Non-Deterministic Composition) *Let $f : T \times T \rightarrow T$ be a commutative and associative operation over T and $p : T \rightarrow$ Bool be a predicate over T such that*

$$p(x) \rightarrow (f(x, y) = x \wedge f(y, x) = x)$$

for all x, y. Then for any expressions b, e and iterators Is

`ndcomp(`f`,` b`,` p`) [` e `|` Is `]`

is deterministic.

We formulate this lemma in a somewhat sloppy fashion by overloading the identifier f with a token in the abstract syntax as well as with its mathematical denotation (a function injected into the domain `Value`).

Corollary 1 (Monotonicity) *Let $(T, \sqsubseteq)$ be totally ordered with maximum m, and $f : T \times T \rightarrow T$ be commutative and associative operation with*

$$x \sqsubseteq f(x, y)$$

for all x, y. Then for any expressions b, e and iterators Is

`ndcomp(`f`,` b`,` $\lambda x.(x$`==`$m)$`) [` e `|` Is `]`

is deterministic.

The corollary states the usual context in which the termination condition is applied (e.g. using $T =$ Bool with true $\sqsubseteq$ false in the function `forall` in Subsection 7.2).

A formal proof of Lemma 16 using the denotational framework would become rather lengthy and not provide much further insight. We therefore commit to the following

Proof Sketch: We define the denotation of non-deterministic compositions as

ES$[\![$`ndcomp(`f`,` b`,` p`) [` e `|` Is`]`$]\!]S :=$ **ES** $($**TC**$[\![$`ndcomp(`f`,` b`,` p`) [` e `|` Is`]`$]\!])S$

Let S be a singleton environment. The denotation

ES$[\![$`ndcomp(`f`,` b`,` p`) [` e `| ]`$]\!]S$

is certainly a singleton. Now consider

ES$[\![$`ndcomp(`f`,` b`,` p`) [` e `|` I `|@`par Is `]`$]\!]S$

By induction we may assume that the definition of `comb` within this denotation yields a deterministic function. The only source of non-determinism are then the applications of `red` in the body of F. We therefore have to show that `red(bag)` is deterministic i.e. that, for any singleton S with $s \in S$,

$$\mathbf{ES}[\![\texttt{red(bag)}]\!]S$$

yields a singleton. Since S is singleton, we have

$$\mathbf{ES}[\![\texttt{bag}]\!]S = \{\text{mkList}(\lambda x.t_1, \text{mkList}(\lambda x.t_2, \ldots \text{mkList}(\lambda x.t_n, mkNil()) \ldots))\}$$

for task results $t_i = \text{mkTuple}_2(f_i, a_i)$. If $f_i = \text{false}$ for all i, then it is easy to see that the denotation of `red(bag)` is

$$\{\text{mkTuple}_2(\text{mkBool}(\text{false}), s([\![\texttt{v}]\!]))\}$$

and we are done. Otherwise we prove that the denotation is

$$\{\text{mkTuple}(\text{true}, f(b_1, f(b_2, \ldots, f(b_{m-1}, b_m))))\}$$

where f is the core denotation of `f` and the b_j are all the a_i with $f_i = \text{mkBool}(\text{true})$ (i.e. there is an injective mapping $g : \mathbb{N}^m \to \mathbb{N}^n$ such that $b_{g(j)} = a_i$ with $f_i = \text{mkBool}(\text{true})$ and for every i with $f_i = \text{mkBool}(\text{true})$ there exists some $j \leq m$ with $g(j) = i$).

Simultaneously we prove that the denotation

$$\mathbf{TS}[\![\texttt{red2(val, bag)}]\!]S$$

yields the singleton

$$\{\text{mkTuple}(\text{true}, f(s([\![val]\!]), f(b_1, \ldots, f(b_{m-1}, b_m)))))\}$$

These proofs proceed by simultaneous induction over the sequence of fixpoint approximations r_i, r'_i of the denotations of `red` and `red2`. The core of the proof is to show that (assuming that the properties are true for r_i and r'_i) in r_{i+1} for any of the possible denotations

$$\text{mkTuple}(t_j, \text{mkList}(\lambda x.t_1, \ldots, \text{mkList}(\lambda x.t_{j-1}, \text{mkList}(\lambda x.t_{j+1}, \ldots)) \ldots))$$

of `any(bag)`, the result is deterministic (with several case distinctions depending on the denotations of `val` and of $p(\texttt{val})$). □

Furthermore, we have to justify the applications of `stop` in `red` and `red2` by showing that their arguments stick to the restrictions defined in Section 5.5.6:

1. Each task in `bag` is nowhere else referenced since all tasks have been created anonymously in F and `red2`.
2. The list `bag` (and each of its tails) is nowhere else referenced, since it is initially passed as `[ ]` to F and in each conditional branch only passed to the recursive invocation (either unmodified or with a new task appended). Likewise `bag` and `bag2` are used in linear form in `red` and `red2`.

Again we omit the formal proof.

7.6.5 A Kernel Extension

The translation of the non-deterministic composition form in Section 7.6 imposes high overhead on the main task that has to perform all non-deterministic synchronizations and start new tasks to perform the required compositions.

In this section, we propose an extension to the PACLIB kernel that is a generalization of on an idea introduced in Qlisp [Goldman and Gabriel, 1988]. In this extension, the same result descriptor serves as the synchronization point of several tasks: when a task terminates, it combines its result with the previous value stored in the descriptor.

Furthermore, we extend the idea such that the compositions can be performed not only sequentially (as suggested by [Goldman and Gabriel, 1988] and sketched in Section 7.6.3) but also in parallel in a tree-like fashion (as in the general translation scheme presented in Section 7.6.2).

In detail, the proposed extension is based on the additional PACLIB functions

```
t = pacGroup(par, c, b, p)
pacCloseFwd(t)
```

where the structure of the kernel task descriptor is redefined as follows:

sem	task	wait	tail	value	*par*	*c*	*p*	buddy

1. The `value` field of the waiting queue is separated into a `tail` field (holding the tail of the waiting task queue) and the actual task result `value`.
2. The `task` field is generalized to a *list* of tasks; `pacWait` checks whether this list is empty to determine whether the result is available yet.
3. `pacStart`, `pacVirtual` and `pacForward` set the *par* field to 0; the semantics of these functions (and of `pacStartFwd` and `pacVirtualFwd`) remains unchanged.
4. `pacGroup` takes the flag *par* (1 or 2), c (the composition function), b (the base value) and p (the termination predicate or NULL). It returns a task descriptor with *par*, c and p correspondingly initialized, the `buddy` flag set to true and `value` set to b.

   Furthermore, a dummy task descriptor is inserted into the wait queue of the descriptor to prevent result delivery before `pacCloseFwd` is called.

   par may be 2 only if c is of type $c : T \times T \rightarrow T$ and thus parallel composition is possible. In case of $c : S \times T \rightarrow T$, *par* must be 1 and the compositions must be performed sequentially on every new value $v : S$.
5. `pacCloseFwd` removes the dummy task from the `task` list and thus allows result delivery. If no other task is in `task`, the final `value` is available and delivered to the blocked tasks in the `wait` queue.
6. The `pacShell` function in every task checks the *par* field of its descriptor when it delivers its task result v.
   (a) If $par = 0$, the task writes its result into `value` and awakes the blocked tasks as described in Section 2.4.
   (b) If $par > 0$ and p is given, the task tests $p(v)$. If the test succeeds, it writes v into `value`, removes its descriptor from the `task` list, aborts the execution of the other tasks in this list, and delivers v to the tasks in the `wait` queue.
   (c) If $par = 1$, the task calls $c(v,$`value`$)$ and updates `value` with the result. During the execution of c, the result descriptor remains locked.
   (d) If $par = 2$ and `buddy` is 'true', the task sets `buddy` to 'false', unlocks the descriptor and continues execution with `c(`v`,value)`.

   If `buddy` is 'false', the task writes v into `value`, sets `buddy` to 'true', removes its descriptor from the `task` list and terminates. If this list is empty, the task delivers v to all tasks blocked in the `wait` queue before.

```
TN⟦h_i:t_i<-l_i, x_j<-a_j..b_j+=c_j, y_k=u_k, defs_1; s; y_k' = r_k, defs_2 |@par R⟧
  ⟦f⟧⟦e⟧⟦v⟧⟦p⟧ =

⟦ let
    auxiliary definitions
    G(h_i, t_i, x_j, y_k) =
      let defs_1; defs_2
      in TN⟦R⟧⟦f⟧⟦e⟧⟦v⟧⟦p⟧
    comb(zs_i, xs_j, ys_k, n) =
      loop( (False, v ) ) [ sum |
        h_i:t_i <- zs_i, x_j <- xs_j .. += c_j, y_k <- rev(zs_k), i <- 1..n,
        sum = (False, v), new = G(h_i, t_i, x_j, y_k),
        next = if sum.0 then
          if new.0 then (True, f(sum.1, new.1)) else sum
          else new;
        ^sum.0 | ^p(sum.1);
        sum' = next ]
    F(z_i, x_j, y_k, zs_i, xs_j, ys_k, n, tasks) =
      case z_i of
        [ ] => startfwd(tasks, comb, zs_i, xs_j, ys_k, n)
        h_i:t_i =>
          if x_j > bv_j
            then startfwd(tasks, comb, zs_i, xs_j, ys_k, n)
            else let
              defs_1
            in if ^s
              then startfwd(tasks, comb, zs_i, xs_j, ys_k, n)
              else let
                defs_2
              in if par
                then F(t_i, x_j+cv_j, r_k, z_i, x_j, [y_k], 1,
                  startfwd(tasks, comb, zs_i, xs_j, ys_k, n))
                else F(t_i, x_j+cv_j, r_k, zs_i, xs_j, y_k:ys_k, n+1, tasks)
    lv_j = l_j; av_j = a_j; bv_j = b_j; cv_j = c_j
  in
    wait(closefwd(F(lv_i, av_j, u_k, lv_i, av_j, [ ], 0, group(2, f, v, p)))) ⟧
```

Figure 7.13: Non-Deterministic Composition (Kernel Extension)

By this extension, we can implement non-deterministic composition in C by the following code skeleton

```
t = pacGroup(2, c, b, p)
while (...) {
  ...
  pacStartFwd(t, ...)
}
pacCloseFwd(t)
r = pacWaitOne(t)
```

where *non-deterministic synchronization is not required at all.* If the *par* flag is 1, value composition is performed sequentially, and in parallel, if the flag is 2.

Assuming that we extend the pseudo-functional language by corresponding operations `group` and `closefwd`, we can implement the non-deterministic composition construct as illustrated in Figure 7.13:

We replace the `tasks` list by a non-deterministic descriptor that is threaded through all invocations of `startfwd` and returned as the result of F: the return value of the task is the result of the combination and can be retrieved by a single application of `wait`.

By this proposal for kernel extension, the results of the individual subtasks do not have an identity on their own; only their combined value can be retrieved by other tasks. In the case of the non-deterministic composition form, however, this is all that we want and need.

The extension has not yet been incorporated into the PACLIB kernel; this will be one of the first tasks of the future (see Chapter 10).

7.7 Analysis

In this section, we analyze the dynamical behavior of the various forms of parallel compositions. We compare them with their sequential counterparts by computing the maximum speedups that can be achieved.

7.7.1 Iterated Composition

We start the time analysis with the basic form of iterated compositions whose behavior essentially captures all other sequential forms:

comp(c,b) [f(x) | x = a; p(x); x' = s(x)]

This form depends on concrete instances for the functions p, s, f, c and the values a and b. The generated imperative target code has the following core structure:

```
F(x) {
  if (p(x)) {
    y = f(x)
    r = F(s(x))
    return(c(y,r)
  }
  else
    return(b);
}
```

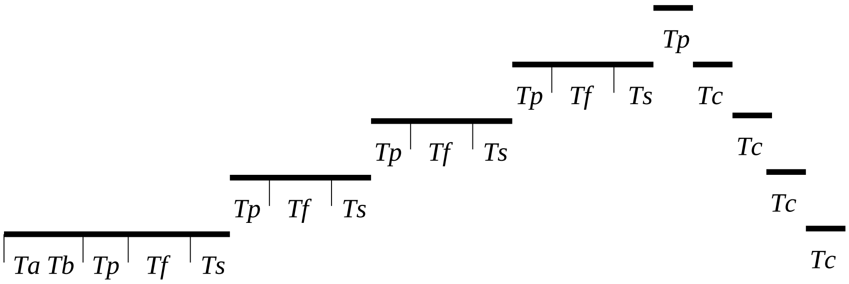


Figure 7.14: Sequential Composition

We assume that the execution times of p, s, f and c are *constant* i.e. do not depend on the concrete values for x. This is of course not true in most programs; we will later discuss generalizations of this simple scheme. The execution time then is (see Figure 7.14)

$$T_{\text{seq}} = T_a + T_b + n(T_p + T_f + T_s + T_c) + T_p.$$

Here $n \geq 0$ is the number of bindings generated for x i.e. the number of iterations.

In the further approximations of the execution times, we will make several simplifying assumptions. First, we will neglect the times required for the evaluation of a and b. This may be justified by the fact that both expressions do not depend on the value for `x` i.e. they can be evaluated *before* the execution of the iteration construct takes place.

Second, we will combine the execution times for s and p to $T_{sp} = T_s + T_p$. This is justified since every application of s is immediately followed by an application of p that tests whether the generated successor value is admissible. We will deliberately ignore the time for the very first test $p(a)$.

The approximated time is then based on the parameters T_{sp}, T_f, and T_c:

- T_{sp} is the essential factor that constrains the speed in which the chain of recursive function invocations is winded off.
- T_f essentially represents the time spent every instance of the invocation chain.
- T_c is the essential factor that constrains the speed in which the chain of recursive function invocations is winded up again.

The approximated execution time then is

$$T_{\text{seq}} \approx n(T_{sp} + T_f + T_c).$$

The definition of T_{seq} holds for `comp` forms as well as for `rcomp` forms (we neglect that `rcomp` iterations are compiled into real loops which are slightly faster than recursive functions).

7.7.2 Technological Parameters

For the further analysis of parallel composition forms we will use the following parameters that depend on the applied technology respectively architecture:

- p is the number of processors.

- T_{start} is the time that a task is actively busy with generating another task i.e. the time until it can continue execution.
- T_{run} is the time between the moment that the parent task is released after the creation of a child task and the moment that the execution of the user-function in the child task begins.
- T_{res} is the time between the moment that the user-function executed by a task returns a value and the time that the task makes this result public such that it can be retrieved by other tasks.
- T_{term} is the time between the moment that a task result has been made public and the moment that the task actually terminates i.e. the time until the processor is free to execute another task.
- T_{wait} is the time that a task requires to determine whether the result of another task is available.
- T_{block} is the time that a task requires to get blocked when it determines that its attempt to retrieve the result of another task has failed.
- T_{awake} ist the time that it takes for a task to deliver its result to a blocked task and to awake that task.

For simplicity, we consider these parameters as constants. However, this is not true in general:

For instance, the creation of a real task by `pacStart` involves the allocation of its workspace from the shared system heap. The execution time of the corresponding system function (which essentially influences T_{start}) may vary very much, in particular if a new virtual memory page is accessed and read from the disk.

Correspondingly, in the PACLIB implementation of virtual tasks, T_{run} is small if some real task inlines a virtual task or the task workspace of a terminated real task is reused by the scheduler for the new task. Otherwise, a new workspace has to be allocated from the shared system heap and T_{run} may become comparatively large.

Furthermore, we will make several assumptions:

1. We assume that the compiler will in the parallel composition forms synchronize task arguments *before* the composer c is executed. In other words, the application of f is the largest subexpression that can be evaluated without synchronization. In general, the compiler will generate better code if it is possible to defer the synchronization into the body of c (see Chapter 6).
2. We assume that tasks are executed in the order of their creation (first-come, first-serve) and are not preemptively descheduled: once executing every task runs without interruption until completion.

   In PACLIB, real tasks are by default preemptively descheduled after fixed time-slices. However, this behavior may be explicitly switched off by the programmer. With virtual tasks, the situation becomes considerably more complicated. Virtual tasks are non-preemptively executed. However, when a task requires the result of a (still virtual) task, it supersedes the scheduler and "inlines" the virtual task. Section 2.6 deals with virtual tasks in more detail.
3. We assume that each of the technological parameters is very small compared to $n(T_{sp} + T_f + T_c)$ (otherwise the parallel composition is much slower than the sequential composition). In the final approximations, we will therefore ignore technological parameters unless there is a large multiplicative factor attached.

4. We will assume that T_f is much larger than T_{sp} (otherwise parallel execution does not pay off). We will therefore always estimate $T_f + \ldots > T_s + T_p + \ldots$ where "…" are sums of technological parameters (without large multiplicative constants).

For the further approximations of execution times, we will fix the technological parameters roughly as they are on our system for (non-virtual) tasks:

- $p = 20$ (Sequent Symmetry with i386@20MHz processors),
- $T_{\text{start}} = 4$ (real task creation, 260 μs),
- $T_{\text{run}} = 1$ (one context switch, 65 μs),
- $T_{\text{res}} = 1$ (roughly as non-blocking wait),
- $T_{\text{term}} = 2$ (deallocation plus one context switch),
- $T_{\text{wait}} = 1$ (non-blocking wait, 60 μs),
- $T_{\text{block}} = 1$ (one context switch),
- $T_{\text{awake}} = 1$ (one context switch).

One time unit in the following sections therefore corresponds on our system to roughly 60μs real time, one second real time are about 16000 time units.

7.7.3 Iterated Task Composition

We consider the iterated composition

```
comp(c,b) [ f(x)@ | x = a; p(x); x' = s(x) ]
```

where not the composition itself but the expression to be composed is annotated. The imperative counterpart has the following structure:

```
 F(x) {
   if (p(x)) {
     yt = start(f,x)
     r = F(s(x))
     return(c(wait(yt),r))
   }
   else
     return(b)
}
```

With this composition form new tasks are started every

$$T_{\text{new}} = T_{sp} + T_{\text{start}}$$

time units and every task has a base "granularity" of

$$T_{\text{task}} = T_{\text{run}} + T_f + T_{\text{res}} + T_{\text{term}}$$

units i.e. during this time a processor is busy with executing the task (neglecting the possible awakening of blocked tasks).

We now have to make a case distinction (see Figure 7.15).

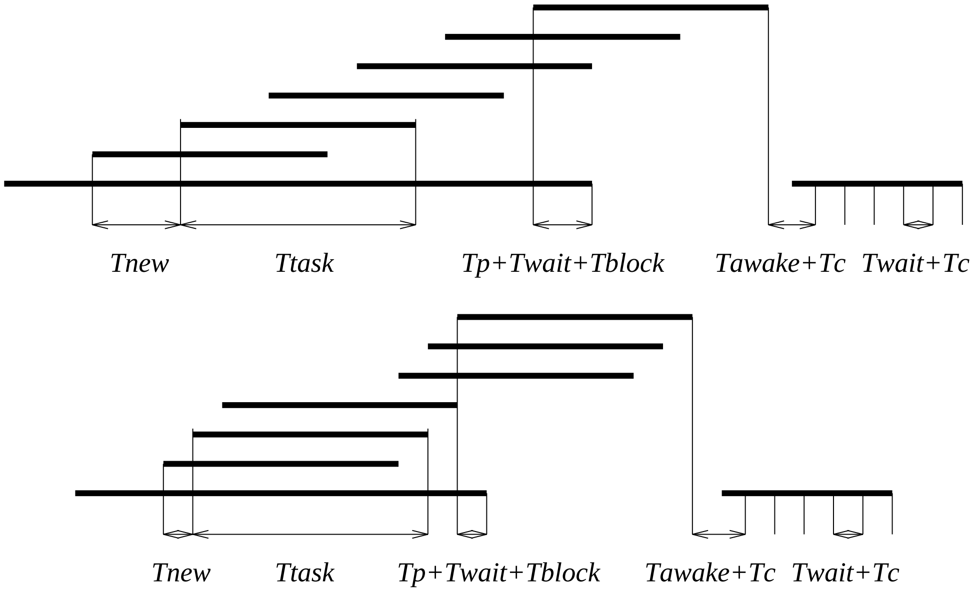


Figure 7.15: Sequential Task Composition ($n = 6, p = 4$)

Case: $n \leq p-1 \ \vee \ T_{\text{task}} \leq (p-1)T_{\text{new}}$

In this case, there are at no time more tasks ready for execution (including the main task) than processors exist or tasks are started at a rate that is so low that every new task immediately finds an idle processor (since its $(p-1)$-th predecessor task has already terminated).

Provided that $T_{\text{run}} + T_f + T_{\text{res}} \geq T_p + T_{\text{wait}} + T_{\text{block}}$, the total runtime is

$$T_{\text{par}} = nT_{\text{new}} + T_{\text{task}} - T_{\text{term}} + T_{\text{awake}} - T_{\text{wait}} + n(T_c + T_{\text{wait}})$$

and after reordering the terms

$$T_{\text{par}} = n(T_{\text{new}} + T_c + T_{\text{wait}}) + T_{\text{task}} + T_{\text{awake}} - T_{\text{term}} - T_{\text{wait}}.$$

Case: $n > p-1 \ \wedge \ T_{\text{task}} > (p-1)T_{\text{new}}$

In this case, tasks are generated so fast that eventually more tasks are ready for execution than processors exist. Only in this situation the the system is saturated and linear speedups can be (at least theoretically) achieved.

Since tasks are started faster than they terminate, partially overlapping "blocks" of p tasks are executed. Since also the main task requires a processor for execution, actually only $p-1$ processors are available for the subtasks until the main task has started the last task and gets blocked.

Let m be the number of blocks of $p-1$ tasks that have been executed before the main task gets blocked i.e.

$$m = \lfloor (nT_{\text{new}} + T_p + T_{\text{wait}} + T_{\text{block}})/T_{\text{task}} \rfloor$$

and consequently

$$m \leq \lfloor (nT_{\text{new}} + T_{sp} + T_{\text{wait}} + T_{\text{block}})/T_{\text{task}} \rfloor$$

In total we have

$$b = m + \lceil (n - m(p-1))/p \rceil = \lceil (n+m)/p \rceil$$

task blocks. The execution time is thus bounded by

$$\begin{array}{rcl} T_{\text{par}} & \leq & bT_{\text{task}} + pT_{\text{new}} \\ & + & n(T_{\text{wait}} + T_c) + T_{\text{awake}} - T_{\text{wait}}. \end{array}$$

We are now going to investigate the achievable speedup i.e. the ratio $T_{\text{seq}}/T_{\text{par}}$. Having fixed the technological constants as discussed in the previous section, T_{sp}, T_f, T_c, and n remain as parameters.

The diagrams in Figure 7.16 show the speedups for varying T_c and T_f and for selected values of n and T_{sp} ($n \in \{19, 99, 499\}$ and $T_{sp} \in \{T_c, 5T_c\}$). The odd values of n where chosen to avoid the case that the last task block consists of a single task only.

The speedup levels of 4, 8, 12, 16, 17, 18 and 19 are displayed as contours on the surfaces. There are several noteworthy observations:

1. The "kick-off" point for the granularity of tasks in order to achieve reasonable efficiency is (for large n) $T_f > 256$ with $T_c < 2.56$.
2. The effect of increasing T_{sp} over T_c (the right set of diagrams) is limited. The zero-height surface in the right bottom corner denotes constellations where the restriction $T_f \gg T_c$ and thus our analysis of the first case is not valid any more.
3. For $n = 19$, the speedup limit is 19. For larger n, the speedup exceeds 19 for $T_f \geq 1024$ and $T_c \leq 0.64$ (the left upper $3 * 3$ square on the surface).
4. For $n = 99$, the surface has discontinuities ("peaks"). This is the border where the surface switches from the (exact) analysis of the first case to the (worst-case) analysis of the second case. The peaks denote the approximation error along this line.
5. For $n = 99$ (and less visible also in $n = 499$), the surface has an "edge" that denotes a constellation of parameters (mainly depending on T_f) where m becomes so low that the number of task blocks b drops by one. This is *not* a consequence of an approximation error but denotes a real-world effect:

   If the last task block just holds a single task, a relatively small increase of T_{task} (by $(p-1)T_{\text{new}}$) suffices to integrate this task into the previous block because the main task now terminates one block earlier. A similar effect is that by increasing the number of tasks n by just 1, the speedup may be reduced if this additional task starts a new task block.

7.7.4 Parallel Composition

We now consider the para-functional expression

```
comp(c,b) [ f(x) | x = a; p(x); x' = s(x) |@ ]
```

with the imperative core structure

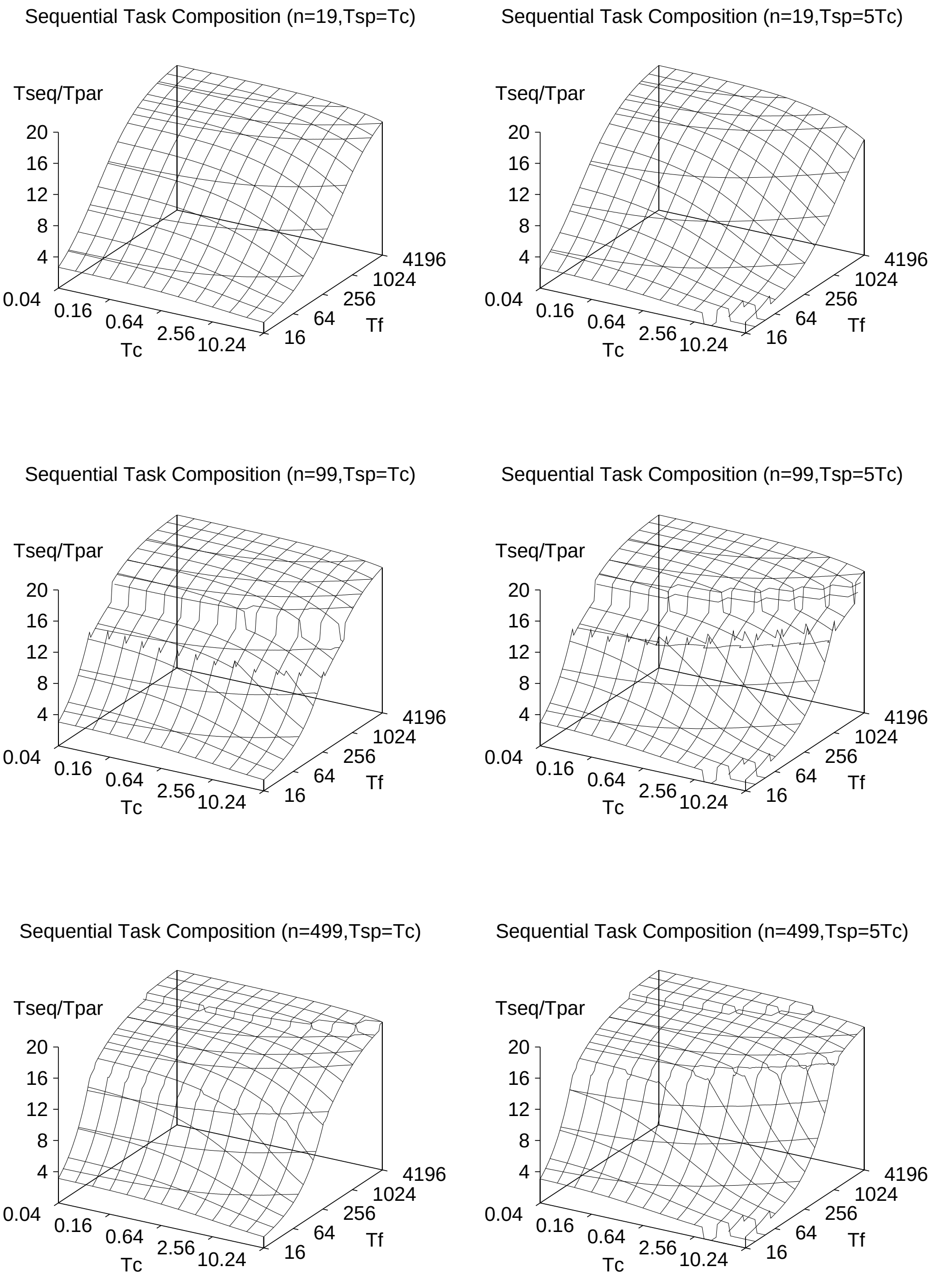


Figure 7.16: Sequential Task Composition Speedups

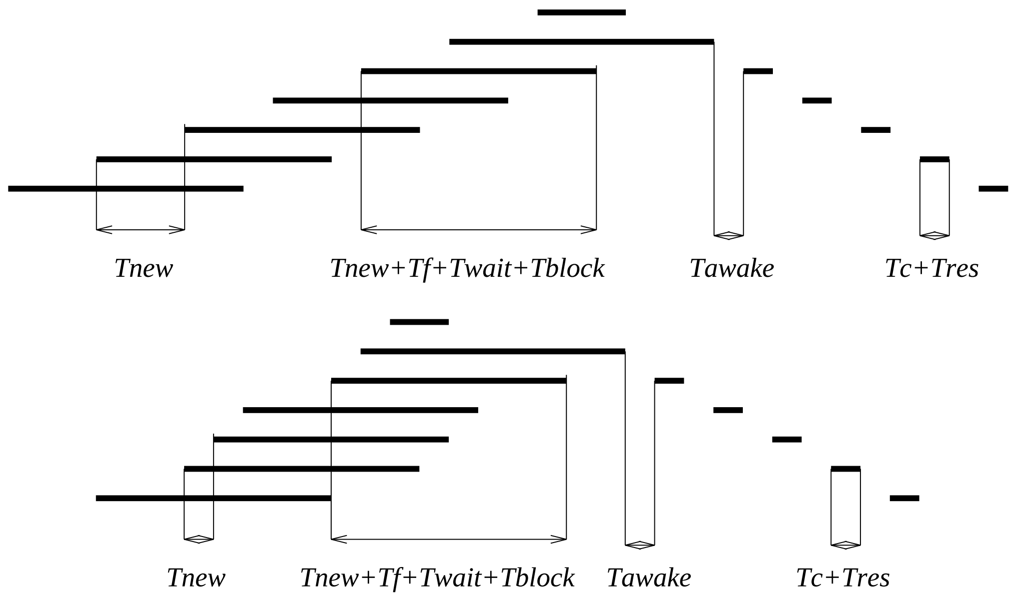


Figure 7.17: Parallel Composition ($n = 6, p = 4$)

```
F(x) {
  if (p(x)) {
    tr = start(F,s(x))
    y = f(x)
    return(c(y, wait(tr)))
  }
  else
    return(b)
}
```

Actually, the new task does not execute F but a function G defined as $G(x) = F(s(x))$ i.e. the successor value is computed within the new task. However, we will neglect this detail.

In this form of parallel composition, new tasks are started every

$$T_{\text{new}} = T_{\text{run}} + T_{sp} + T_{\text{start}}$$

time units because every task is started by its predecessor task, therefore T_{new} includes the execution delay T_{run}. Every task has a granularity of

$$T_{\text{task}} = T_{\text{new}} + T_f + T_{\text{wait}} + T_{\text{block}} + T_c + T_{\text{res}} + T_{\text{awake}} + T_{\text{term}}$$

since it is blocked on the attempt to receive the result from its child task (see Figure 7.17).

Case: $n \leq p \;\vee\; T_{\text{task}} \leq pT_{\text{new}}$

Assuming that $T_f + T_{\text{wait}} > T_{\text{run}} + T_{sp} + T_{\text{res}}$, every task finds a free processor and the total runtime is for $n > 0$

$$\begin{aligned} T_{\text{comp}} &= T_p + T_{\text{start}} + (n-1)T_{\text{new}} + T_f + T_{\text{wait}} \\ &+ (n-1)(T_c + T_{\text{res}} + T_{\text{awake}}) + Tc \end{aligned}$$

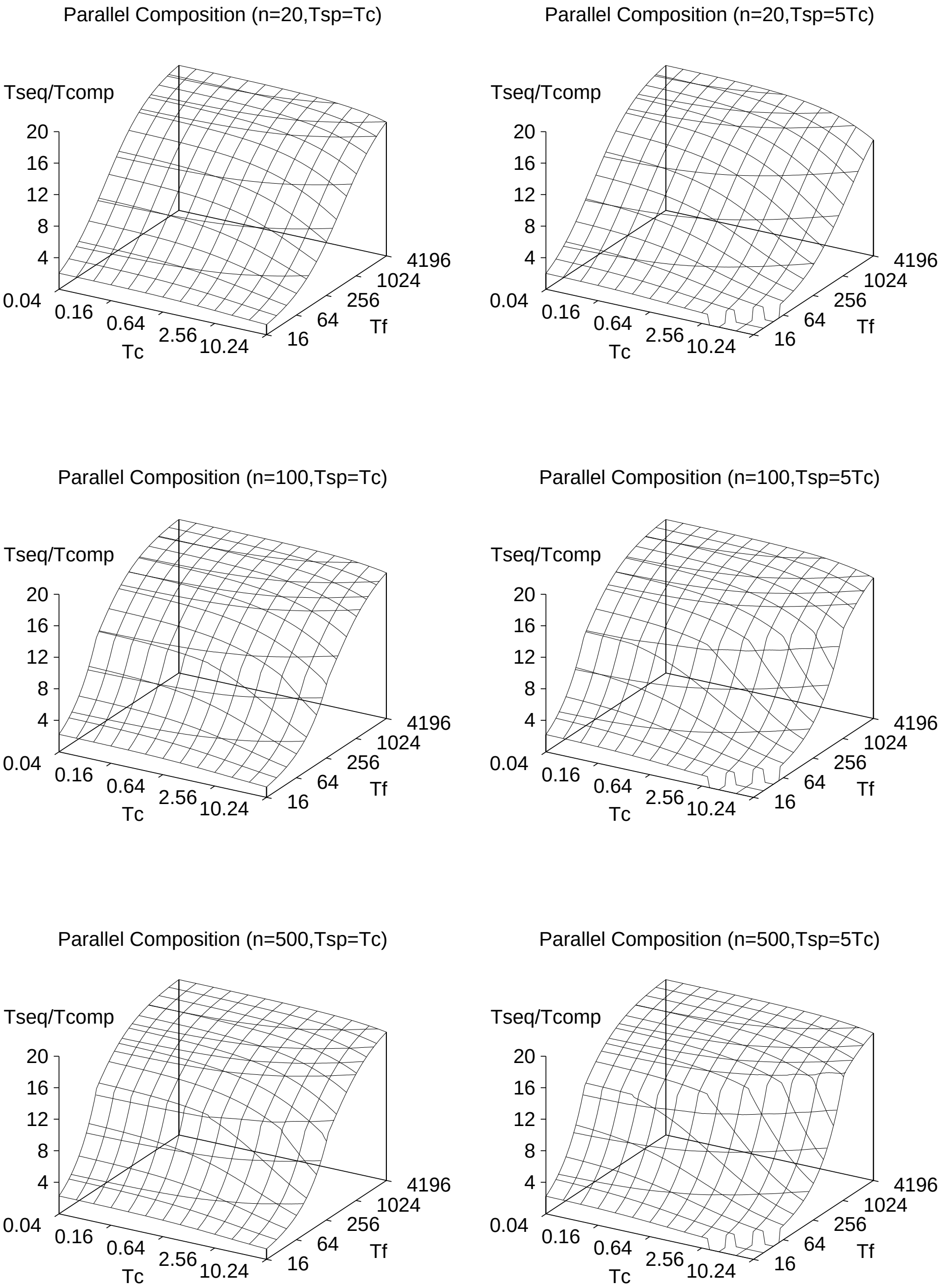


Figure 7.18: Parallel Composition Speedups

which we approximate by

$$T_{\text{comp}} \approx n(T_{sp} + T_c + T_{\text{start}} + T_{\text{run}} + T_{\text{res}} + T_{\text{awake}}) + T_f\,.$$

Case: $n > p \;\wedge\; T_{\text{task}} > pT_{\text{new}}$

Assuming that $T_f + T_{\text{wait}} > T_{\text{run}} + T_{sp} + T_{\text{res}}$ the total runtime is for $n > 0$

$$\begin{aligned} T_{\text{comp}} &= T_p + \lceil n/p\rceil(T_{\text{new}} + T_f + T_{\text{wait}} + T_{\text{block}}) \\ &+ (n+p-1 \bmod p)T_{\text{new}} + (n-1)(T_c + T_{\text{res}} + T_{\text{awake}}) + T_c \end{aligned}$$

which we approximate by

$$\begin{aligned} T_{\text{comp}} &\leq \lceil n/p\rceil(T_{sp} + T_f + T_{\text{start}} + T_{\text{run}} + T_{\text{wait}} + T_{\text{block}}) \\ &+ pT_{\text{new}} + n(T_c + T_{\text{res}} + T_{\text{awake}}) \end{aligned}$$

Compared to the diagrams in the previous subsection, we notice for the speedup surfaces in Figure 7.18:

1. The surfaces are smoother. There are no peaks and no edges, because there is no distinct "main" task and hence all all tasks perform exactly the same actions. Consequently, the number of executed task blocks $\lceil n/p\rceil$ is independent of T_f and T_{sp}.
2. The speedup is considerably reduced for small T_c and small T_f ($T_c < 2.56$ and $T_f < 256$). This is a consequence of the additional delay in the creation of new tasks (the factor T_{run}) and in the combination of task results (the factor $T_{\text{res}} + T_{\text{awake}}$). For large n ($n = 499$) however this difference becomes insignificant.

7.7.5 Reverse Composition

We now consider the form

rcomp(c,b) [f(x) | x = a; p(x); x' = s(x) |@]

where the values are composed in the order of their generation:

```
F(x,rt) {
  while (p(x)) {
    rt = start(C,x,rt)
    x = s(x)
  }
  return(wait(rt))
}
C(x,rt) {
  y = f(x)
  return(c(y,wait(x)))
}
```

All tasks are started by the main task every

$$T_{\text{new}} = T_{sp} + T_{\text{start}}$$

time units. The total behavior of the composition depends very much on whether a task receives the result of its predecessor task with or without blocking. For simplicity, we

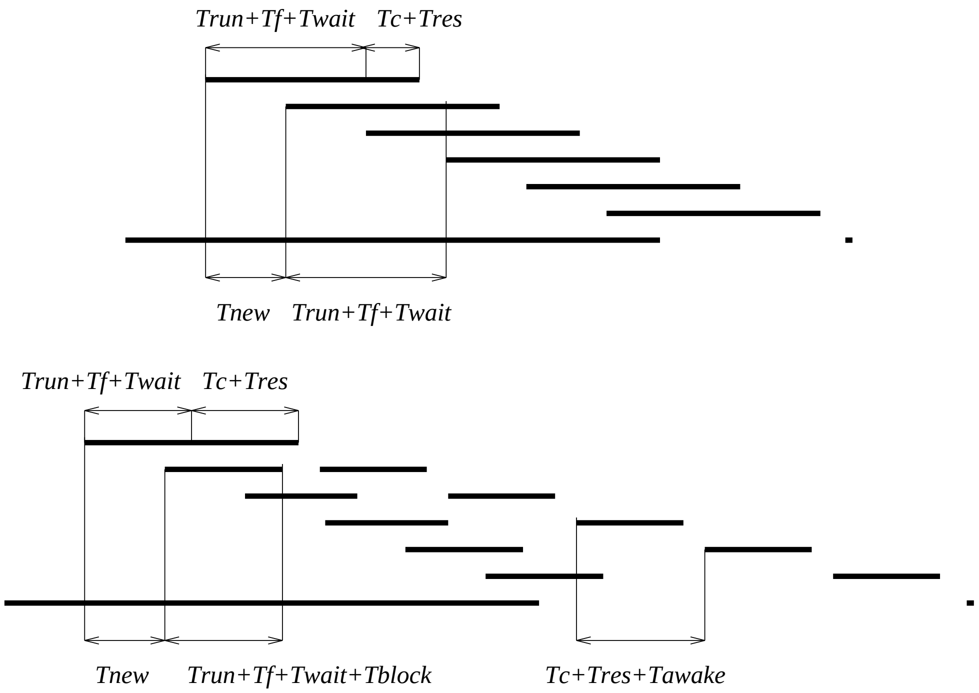


Figure 7.19: Reverse Composition (Non-Blocking and Blocking)

assume that the base value b is also task handle such that every task executes the same code (actually, the first task receives the non-task b and thus saves one `wait` operation).

The first task does not block on b and takes time

$$T_0 = T_{\text{run}} + T_f + T_{\text{wait}} + T_c + T_{\text{res}}$$

until its result is delivered. The successor task takes time

$$T_1 = T_{\text{new}} + T_{\text{run}} + T_f + T_{\text{wait}}$$

until it requests the result of its predecessor. We now have the two cases (see Figure 7.19):

1. If $T_0 \leq T_1$, i.e. $T_c + T_{\text{res}} \leq T_{\text{new}}$, the task is not blocked and delivers its result at time

   $$T_2 = T_1 + T_c + T_{\text{res}} = T_{\text{new}} + T_0.$$

   By induction, every further task receives the predecessor result without blocking and has granularity

   $$T_{\text{task}} = T_0 + T_{\text{term}} = T_{\text{run}} + T_f + T_{\text{wait}} + T_c + T_{\text{res}} + T_{\text{term}}.$$

2. If $T_0 > T_1$, i.e. $T_c + T_{\text{res}} > T_{\text{new}}$, the successor task is blocked and delivers its result at time

   $$T_2 = T_0 + T_{\text{awake}} + T_c + T_{\text{res}}$$

   By induction, every further task is also blocked and takes until blocking time

   $$T_{\text{task}} = T_{\text{run}} + T_f + T_{\text{wait}} + T_{\text{block}}.$$

We now make a case distinction (see Figure 7.20).

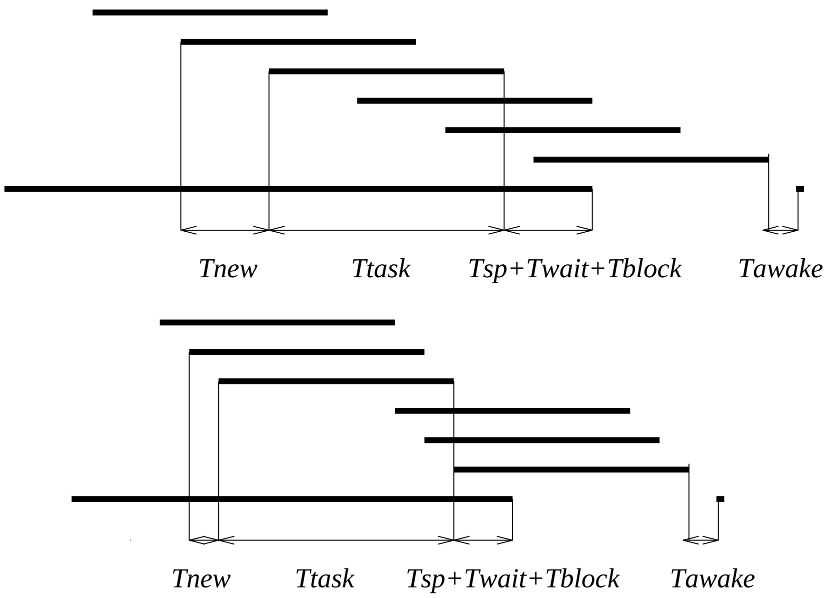


Figure 7.20: Reverse Composition ($n = 6, p = 4$)

Case: $n \leq p - 1 \;\vee\; T_{\text{task}} \leq (p-1)T_{\text{new}}$

In this case every task finds an idle processor.

In the non-blocking case ($T_c + T_{\text{res}} \leq T_{\text{new}}$), the total runtime is

$$T_{\text{rcomp}} = T_p + T_{\text{start}} + (n-1)T_{\text{new}} + T_{\text{task}} - T_{\text{term}} + T_{\text{awake}}.$$

which we approximate by

$$T_{\text{rcomp}} \approx n(T_{sp} + T_{\text{start}}) + T_f + T_c.$$

In the blocking case ($T_c + T_{\text{res}} > T_{\text{new}}$), the total runtime is

$$T_{\text{rcomp}} = T_p + T_{\text{start}} + T_{\text{run}} + T_f + T_{\text{wait}} + n(T_c + T_{\text{res}} + T_{\text{awake}})$$

which we approximate by

$$T_{\text{rcomp}} \approx n(T_c + T_{\text{res}} + T_{\text{awake}}) + T_{sp} + T_f.$$

Case: $n > p - 1 \;\wedge\; T_{\text{task}} > (p-1)T_{\text{new}}$

In this case, eventually more tasks are ready for execution than processors exist. Furthermore, since the main task generates all other tasks, only $p-1$ processors are available until the main task is blocked for the result.

Let m be the number of task blocks executed before the main task gets blocked i.e.

$$m = \lfloor (nT_{\text{new}} + T_{sp} + T_{\text{wait}} + T_{\text{block}})/T_{\text{task}} \rfloor$$

In the non-blocking case ($T_c + T_{\text{res}} \leq T_{\text{new}}$), the total runtime is bounded by

$$T_{\text{rcomp}} < \lceil (n+m)/p \rceil T_{\text{task}} + pT_{\text{new}} + T_{\text{awake}}$$

which we approximate by

$$T_{\text{rcomp}} \approx \lceil (n+m)/p \rceil (T_f + T_c + T_{\text{run}} + T_{\text{wait}} + T_{\text{res}} + T_{\text{term}}) + p(T_{sp} + T_{\text{start}})$$

In the blocking case ($T_c + T_{\text{res}} > T_{\text{new}}$), it is much more difficult to anticipate the exact scheduling order since the main task may still start new tasks when the already executing tasks are blocked and awaked again.

The optimal scheduling strategy is to insert all awaked tasks at the beginning of the ready queue such that they are executed before the tasks that have not yet executed at all. Then the combinations can be performed in parallel with the evaluations of other task. A similar effect can be also achieved with preemptive time-slicing.

However, since we assume non-preemptive FIFO scheduling, the situation is different. In the worst case, all tasks have already been started when the first task is blocked. In this case, all tasks get blocked before the first task is awaked again to perform a combination step; hence during the combination phase only one task is active at a time.

In general, the question is what is the number q of not yet executed tasks in the ready queue when the last task has been started. When these tasks get eventually blocked, they are inserted at the tail of the queue and in the end q combinations have to be performed in sequence.

Since the time from the creation of the first to the creation of the last task is nT_{new} and every task takes until blocking time T_{task} we can give the (rather pessimistic) bound

$$q \leq n - \lfloor (n-1)T_{\text{new}}/T_{\text{task}} \rfloor$$

The total runtime is bounded by

$$\begin{aligned} T_{\text{rcomp}} &\leq \lceil (n+m)/p \rceil T_{\text{task}} \\ &+ pT_{\text{new}} + q(T_c + T_{\text{res}} + T_{\text{awake}}) + T_{\text{awake}} \end{aligned}$$

which we approximate by

$$\begin{aligned} T_{\text{rcomp}} &\approx \lceil (n+m)/p \rceil (T_f + T_{\text{run}} + T_{\text{wait}} + T_{\text{block}}) \\ &+ p(T_{sp} + T_{\text{start}}) + q(T_c + T_{\text{res}} + T_{\text{awake}}). \end{aligned}$$

1. The diagrams in Figure 7.21 show the non-blocking case where the composition time T_c is not longer than T_{sp}. Compared to the previous sections, the speedups are drastically improved even for small T_f and large T_c. The main reason for this is that the composition time is fully subsumed by the time for task creation.

2. In Figure 7.22, however we see the effects of the blocking case if T_c is already large and T_{sp} is even significantly larger: We observe a "surface fold" at $T_c \approx 5$ i.e. along this line the speedup is abruptly reduced.

   This fold denotes the border where the blocking of tasks sets in and many compositions have to be performed sequentially in the end. Please note, however, that our bound for q is rather pessimistic, the effect may be less dramatical in reality. For large T_f, the difference becomes less significant.

7.7.6 Tree Composition

We now consider the form

```
tcomp(c,b) [ f(x) | x = a, i <- 1.. ; p(x); x' = s(x) |@(i%r==0) ]
```

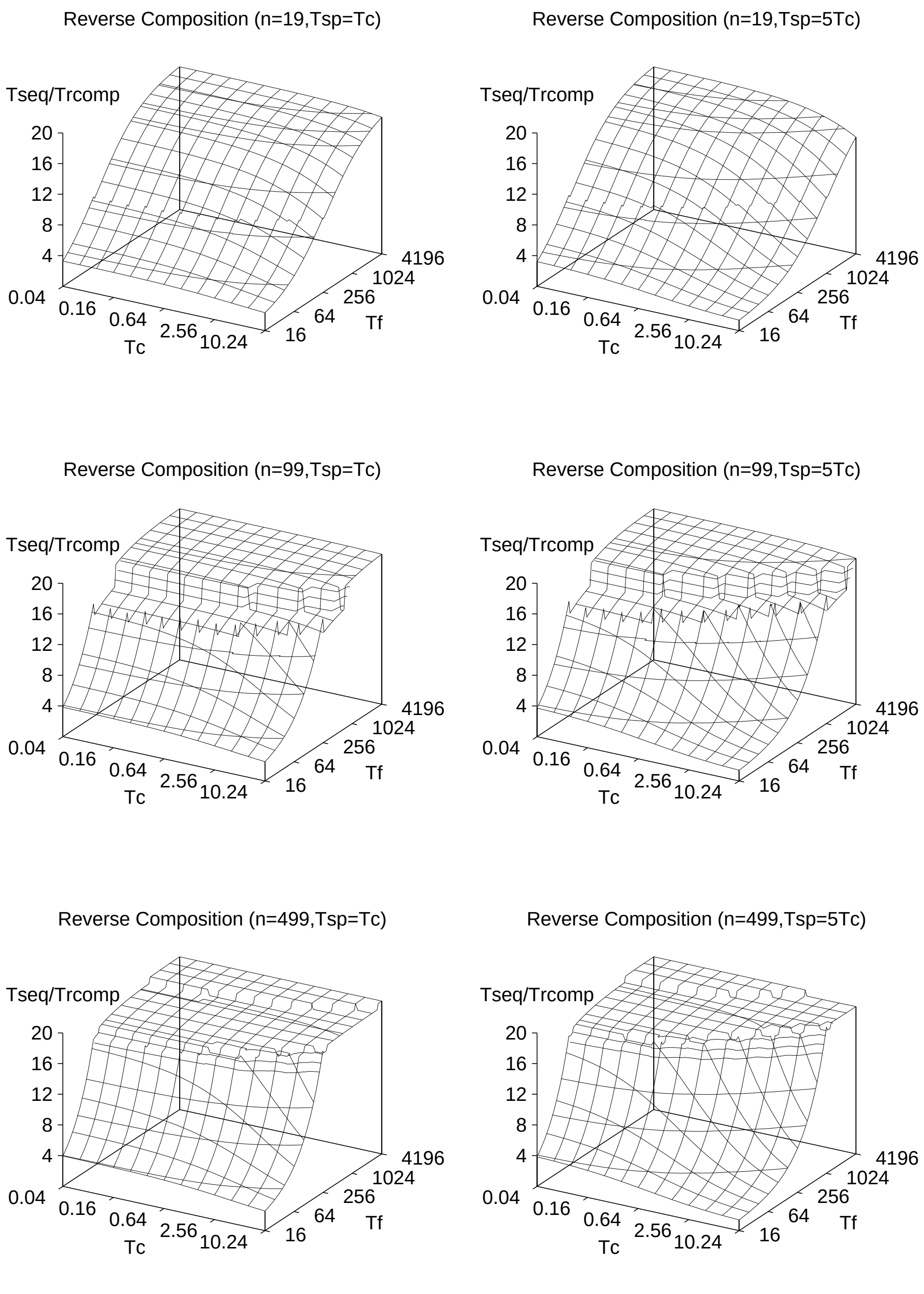


Figure 7.21: Reverse Composition Speedups (Non-Blocking)

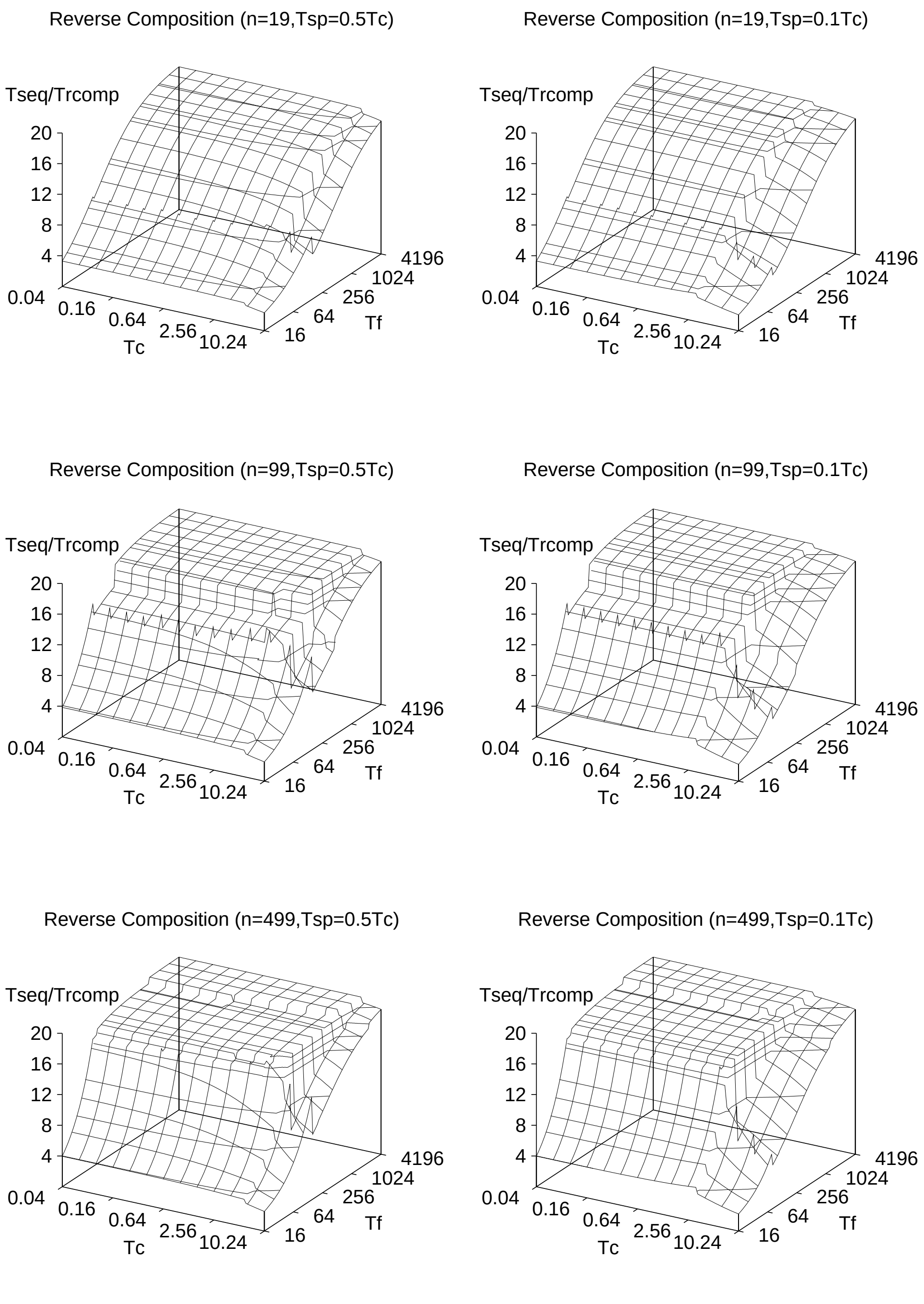


Figure 7.22: Reverse Composition Speedups (Blocking)

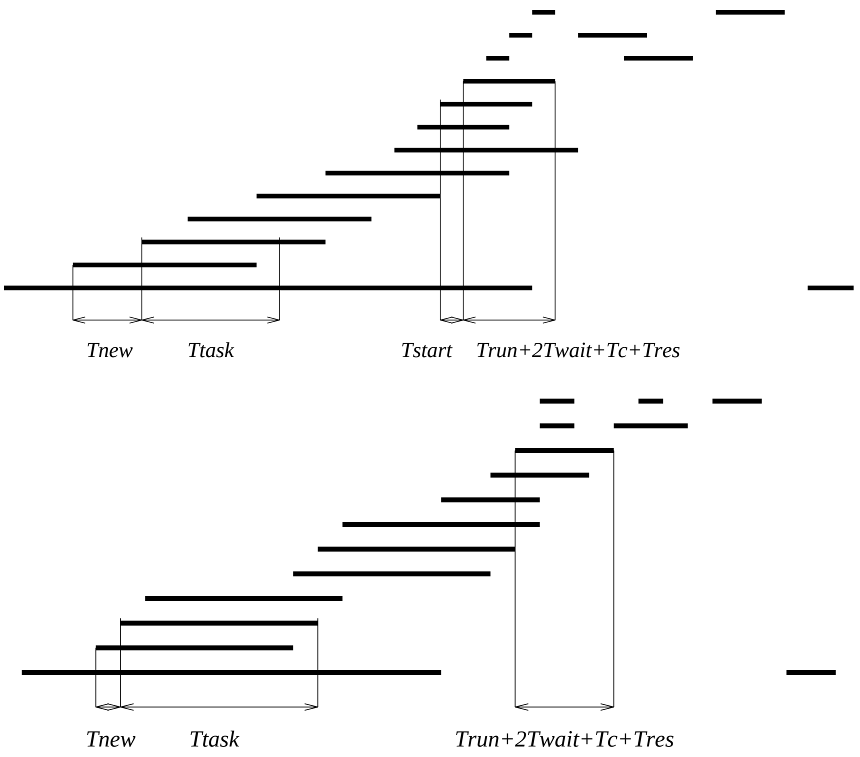


Figure 7.23: Tree Composition ($q = 6, p = 4$)

where the values are composed in the fashion of a binary tree. In this composition form, the programmer has the possibility to control the number of bindings processed by every task (and thus the number of tasks):

```
F(x,i) {
  tl = nil
  xl = nil
  while (p(x)) {
    xl = cons(x,xl)
    if (i%r == 0) {
      t = start(F0, xl)
      tl = cons(t, tl)
      xl = nil
    }
    i = i+1
    x = s(x)
  }
  return(c(reduce(tl),b))
}
```

The function `reduce` traverses the task list *without* synchronizing the individual tasks and creates new tasks to perform the synchronizations and combinations in the form of a binary tree. The last task returns the total result.

Let r be the number of bindings per task; then $q = \lceil n/r \rceil$ tasks are created. These q tasks are started by the main task every

$$T_{\text{new}} = rT_{sp} + T_{\text{start}}$$

time units and every task (but the last one) has granularity

$$T_{\text{task}} = T_{\text{run}} + r(T_f + T_c) + T_{\text{res}} + T_{\text{term}}.$$

(neglecting possible awakening of other tasks). In addition, another $q-1$ tasks are started every T_{start} units to perform the final compositions in the form of a binary tree. Each of these tasks has a granularity of

$$T_{\text{task2}} = T_{\text{run}} + 2T_{\text{wait}} + T_c + T_{\text{res}} + T_{\text{term}}.$$

(neglecting possible blocking).

Case: $q \leq p-1 \;\vee\; T_{\text{task}} \leq (p-1)T_{\text{new}}$

In this case, the first q tasks find idle processors. However, this need not be true for the additional $q-1$ composition tasks since these tasks are started in smaller distances (T_{start}) and have other granularities (T_{task2}).

However, if $q+1 \leq p$, all tasks find idle processors. The total runtime is then determined by the critical path that consists of the last of the q tasks and the compositions that depend on its result i.e.

$$\begin{array}{rcl} T_{\text{tcomp}} & \approx & nT_{sp} + 2qT_{\text{start}} + T_f \\ & + & \lceil 1 + \log_2 q \rceil (T_c + T_{\text{res}} + T_{\text{awake}}). \end{array}$$

Otherwise, if $q+1 > p$, the composition tasks need not find idle processors but have to wait until the previous tasks (and the main task) have terminated. They are then executed in blocks of p tasks. When the width of the composition tree falls below the number of processor, the tasks are blocked and awaked again. The runtime is thus approximated by

$$\begin{array}{rcl} T_{\text{tcomp}} & \approx & nT_{sp} + 2qT_{\text{start}} + T_f + pT_{\text{new}} \\ & + & \lceil (q - 2^{\lfloor 1+\log_2 p \rfloor})/p \rceil (T_{\text{run}} + 2T_{\text{wait}} + T_c + T_{\text{res}} + T_{\text{term}}) \\ & + & \lfloor 1 + \log_2 p \rfloor (T_c + T_{\text{res}} + T_{\text{awake}}). \end{array}$$

Case: $q > p-1 \;\wedge\; T_{\text{task}} > (p-1)T_{\text{new}}$

In this case, the first q tasks saturate the system. By the first-come/first-serve scheduling, these tasks are executed and deliver their result before the corresponding $q-1$ combination tasks are executed; thus the first combination tasks are not blocked for these results. However, at the end, when the number of combination tasks falls below the number of processors, tasks are blocked and have to be awaked again.

Let m be the number of blocks of $p-1$ tasks executed until the main task gets blocked i.e.

$$m \leq \lfloor ((n+1)T_{sp} + (2q-1)T_{\text{start}} + T_{\text{wait}} + T_{\text{block}})/T_{\text{task}} \rfloor$$

Then the total runtime is bounded by

$$\begin{array}{rcl} T_{\text{tcomp}} & \leq & \lceil (q+m)/p \rceil T_{\text{task}} + pT_{\text{new}} \\ & + & \lceil (q - 2^{\lfloor 1+\log_2 p \rfloor})/p \rceil T_{\text{task2}} \\ & + & \lfloor 1 + \log_2 p \rfloor (T_{\text{task2}} + 2T_{\text{block}} + T_{\text{awake}} - T_{\text{term}}) + T_c \end{array}$$

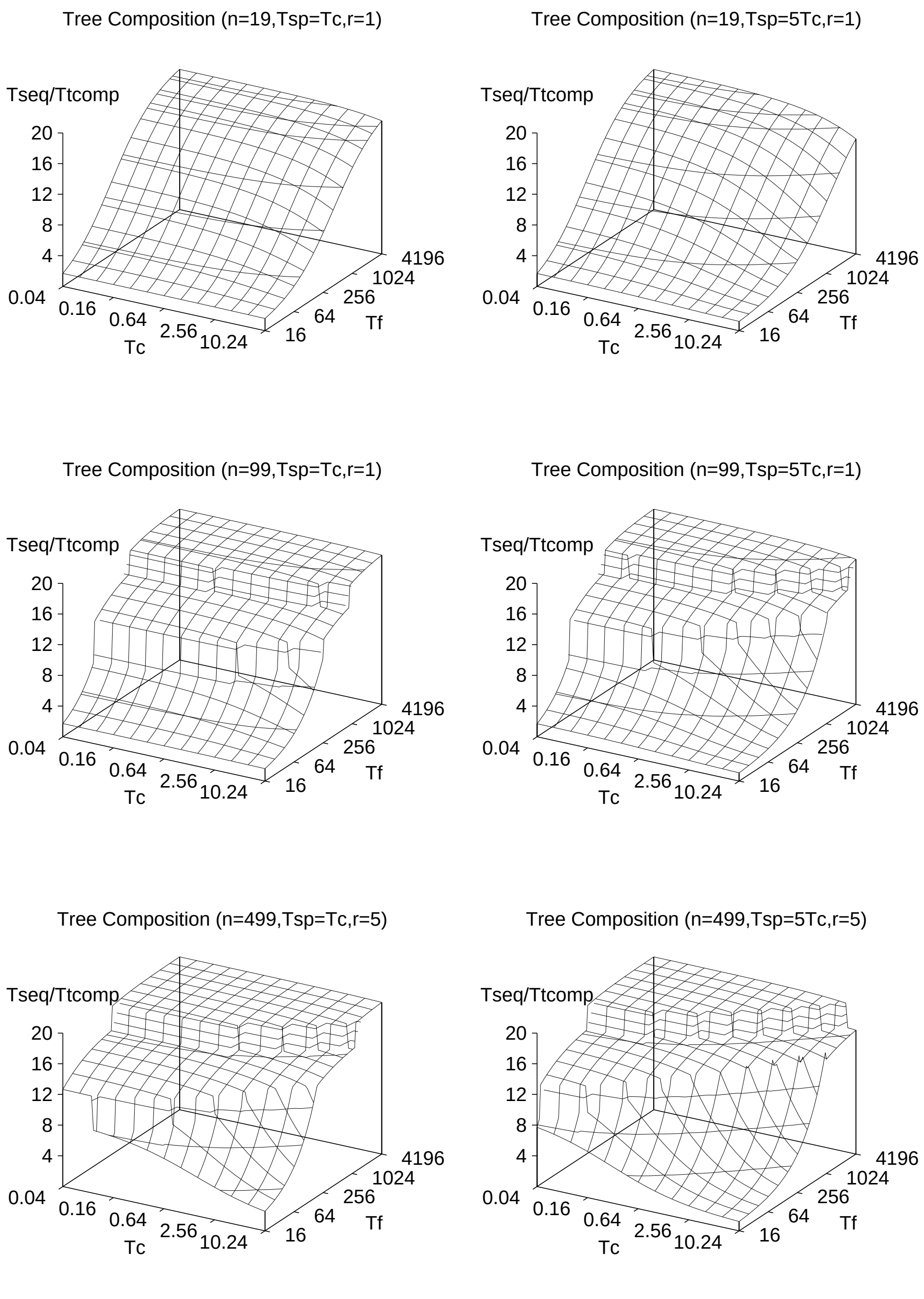


Figure 7.24: Tree Composition Speedups

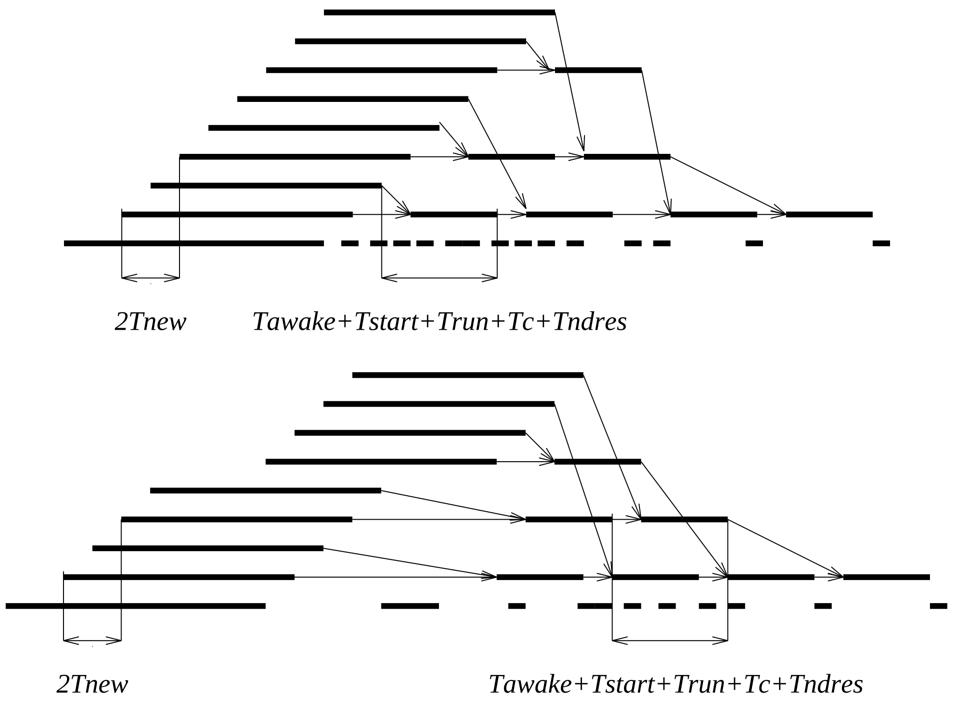


Figure 7.25: Non-Deterministic Composition ($q = 8, p = 5$)

which we approximate by

$$\begin{aligned} T_{\text{tcomp}} &\approx \lceil (q+m)/p \rceil (r(T_f + T_c) + T_{\text{run}} + T_{\text{res}} + T_{\text{term}}) \\ &+ p(rT_{sp} + T_{\text{start}}) \\ &+ \lceil (q - 2^{\lfloor 1+\log_2 p \rfloor})/p \rceil (T_c + T_{\text{run}} + 2T_{\text{wait}} + T_{\text{res}} + T_{\text{term}}) \\ &+ \lfloor 1 + \log_2 p \rfloor (T_c + T_{\text{run}} + 2T_{\text{wait}} + 2T_{\text{block}} + T_{\text{res}} + T_{\text{awake}}) + T_c. \end{aligned}$$

The results are displayed in Figure 7.24:

1. For larger n the speedups are drastically improved and reach the hardware "plateau" for a wider range of T_f and T_c. This is mainly a consequence of the fact that the combinations are performed in parallel.
2. However, for very small T_c and $r = 1$, the speedup is worse than in the corresponding diagrams in Figure 7.16. This is because of the overhead for the additional composition tasks that does not pay off in this case.

7.7.7 Non-Deterministic Composition

Finally, we consider the form

```
ndcomp(c,b) [ f(x) | x = a, i <- 1.. ; p(x); x' = s(x) |@(i%r==0) ]
```

where the generated values are composed in a non-deterministic order. This construct behaves like `tcomp` except for the composition phase. Here the main task performs the synchronizations and starts a composition task for every pair of results:

```
while (q > 1) {
  (a,tl) = any(tl)
  (b,tl) = any(tl)
  t = start(c, a, b)
  tl = cons(t, tl)
  q = q-1
}
return(wait(head(l)))
```

Every original task processes r bindings. The main task starts $q = \lceil n/r \rceil$ tasks every

$$T_{\text{new}} = rT_{sp} + T_{\text{start}}$$

time units; every task (but the last one) has granularity

$$T_{\text{task}} = T_{\text{run}} + r(T_f + T_c) + T_{\text{res}} + T_{\text{term}}.$$

In addition, another $q-1$ tasks are started to perform the final compositions. Each of these tasks has a granularity of

$$T_{\text{task2}} = T_{\text{run}} + T_c + T_{\text{res}} + T_{\text{term}}.$$

(neglecting a possible awakening of the main task).

Case: $q \leq p-1 \ \vee \ T_{\text{task}} \leq (p-1)T_{\text{new}}$

In this case, each of the initial q tasks immediately finds an idle processor.

All synchronizations are performed by the main task; the critical path consists of the first terminated task and the the corresponding sequence of composition operations. To see this, take a look at Figure 7.25 with the two sets of $q = 8$ tasks and $q-1 = 7$ composition tasks shown side by side.

The main task performs the first non-deterministic wait operation at

$$T_0 \approx nT_{sp} + qT_{\text{start}}.$$

At this time, t tasks have already terminated with

$$t \leq \lfloor (\max\{T_0, T_{\text{task}}\} - T_{\text{task}})/T_{\text{new}} \rfloor.$$

For these t tasks, $u = \lfloor t/2 \rfloor$ composing tasks have to be started. The time T_1 at which the main task gets blocked is therefore

$$T_1 \leq T_0 + uT_{\text{start}} + (t+1)T_{\text{ndwait}} + T_{\text{block}}.$$

How many of these composition tasks are executed in parallel?

The q tasks deliver a result every T_{new} time units, consequently every $2T_{\text{new}}$ units a composition task may be started. However the main task needs at least time $2T_{\text{ndwait}} + T_{\text{start}}$ to receive the results of two tasks and start another task. Therefore a composition task can be started at most every T_{main} units with

$$T_{main} = \max\{2T_{\text{new}}, 2T_{\text{ndwait}} + T_{\text{start}}\}.$$

The synchronization time T_{ndwait} is linear in the number of tasks q that is waited for. More exactly, it is proportional to the ratio $q/(1+t)$ (where t is the number of terminated tasks) because that many tasks have to be investigated in average to find a terminated task.

We estimate the time complexity for the first non-deterministic wait operation therefore as $q/(1+t)T_{\text{wait}}$. During the subsequent t waits, the denominator decreases by 1 every wait, and the numerator decreases by 1 every second wait (for every two terminated tasks a new task is started). Assuming that no task terminates, finally the ratio $q - t + u \approx q - t/2$ is reached. We therefore have a total synchronization complexity of about

$$\begin{aligned} S_0 &\approx \int_0^t \frac{q-i/2}{1+t-i} di \\ &= \int_0^t 1/2 + \frac{q-\frac{1+t}{2}}{1+t-i} di \\ &= 1/2(i-(2q-t))\ln(1+t-i))|_0^{t-1} \\ &= 1/2(t+(2q-t)\ln(1+t)) \end{aligned}$$

and thus

$$T_1 \approx T_0 + uT_{\text{start}} + S_0 T_{\text{wait}} + T_{\text{block}}.$$

Consequently we estimate in T_{main}

$$T_{\text{ndwait}\,0} \approx S_0/(2t+1)T_{\text{wait}}$$

The number of composition tasks that may execute in parallel is now w with

$$w = \min\{p-1, \lceil T_{\text{task2}}/T_{main} \rceil\}$$

i.e. the tree of combination tasks has a maximum "width" of w. In this estimation, a processor is entirely reserved for the main task to perform the necessary synchronizations, which is a worst-case assumption.

The "height" of the composition tree (the number of tasks in the critical path) is therefore $h = h_0 + h_1 + h_2$ where

$$\begin{aligned} h_0 &= \lceil u/w \rceil \\ h_1 &= \lceil (q-u-2w+1)/w \rceil \\ h_2 &= \lceil 1 + \log_2 w \rceil \end{aligned}$$

(if $q - u < 2w$, then $h_1 = 0$).

The first h_0 levels represent the phase during which those tasks are composed that have terminated before T_0. The next h_1 levels represent the execution of the composition tasks started after T_1 when still more composition tasks than processors exist. In the remaining h_2 levels, there are fewer tasks active than processors exist.

The minimum distance between the result delivery of a task and the execution of the succeeding composition task is at least $T_{\text{ndwait}} + T_{\text{start}}$. During the first h_0 levels, $T_{\text{ndwait}\,0}$ is estimated as described above. In the next h_1 levels, there are initially $q - u$ composition tasks all of which may be executing. The total number of tasks in the subsequent waits is therefore

$$S_1 = (q-u) + 2(q-u-1) + 2(q-u-2) + \ldots + w = (q-u)^2 - w^2$$

i.e. we estimate

$$T_{\text{ndwait}\,1} \approx S_1/(2(q-u-w)+1)T_{\text{wait}} \approx (q-u+w)/2T_{\text{wait}}$$

In the remaining h_1 levels, there are initially w tasks. The total number of tasks waited for is therefore

$$S_2 = w + (w-1) + (w-1) + \ldots + 1 + 1 = w^2$$

and we estimate

$$T_{\text{ndwait}\,2} \approx S_1/(2w+1)T_{\text{wait}} \approx w/2T_{\text{wait}}$$

We approximate now the total runtime as

$$\begin{aligned} T_{\text{ndcomp}} &\approx nT_{sp} + qT_{\text{start}} + T_f \\ &+ h_0(T_{\text{run}} + T_c + T_{\text{res}} + T_{\text{ndwait}\,0} + T_{\text{start}}) \\ &+ h_1(T_{\text{run}} + T_c + T_{\text{res}} + T_{\text{ndwait}\,1} + T_{\text{start}}) \\ &+ h_2(T_{\text{run}} + T_c + T_{\text{res}} + T_{\text{ndwait}\,2} + T_{\text{start}}). \end{aligned}$$

Case: $q > p-1 \;\wedge\; T_{\text{task}} > (p-1)T_{\text{new}}$

Here blocks of $p-1$ tasks are executed until the main task gets blocked in a non-deterministic wait. Because of the limited number of processors, the main task gets the next chance for execution only after the termination of the initial q tasks and of the already started composition tasks.

The main task performs its first non-deterministic wait operation at time

$$T_0 = nT_{sp} + qT_{\text{start}} + T_p.$$

At this time, m blocks of $p-1$ tasks have terminated with

$$m = \lfloor T_0/T_{\text{task}} \rfloor$$

i.e. the number of terminated tasks is $t = (p-1)m$. For these t tasks, $u = \lfloor t/2 \rfloor$ composing tasks have to be started. The time T_1 at which The main task gets blocked is therefore

$$T_1 \approx T_0 + uT_{\text{start}} + S_0 T_{\text{wait}} + T_{\text{block}}.$$

where S_0 is estimated as described in the previous subsection.

After the main task is awaked, it has to start the remaining composition tasks to combine the remaining $q-t+u$ tasks most of which have already terminated (the last $p-1$ tasks may be still executing).

The main task is able to start a new task at most every $2T_{\text{ndwait}} + T_{\text{start}}$ time units; the width w of the composition tree is therefore limited by

$$w = \lceil \min\{p-1, T_{\text{task2}}/(2T_{\text{ndwait}} + T_{\text{start}})\} \rceil$$

i.e. at most w tasks can execute in parallel. Unlike in the previous subsection, this term does *not* depend on T_{new}, because the initial q tasks have already terminated when the main task is awaked again. Again we use a conservative bound $p-1$ for the number of processors.

The rest of the analysis proceeds like in the previous subsection.

The total runtime is therefore approximated by

$$\begin{aligned} T_{\text{ndcomp}} &\approx \lceil (m+q)/p \rceil T_{\text{task}} + \\ &+ h_0(T_{\text{run}} + T_c + T_{\text{res}} + T_{\text{ndwait}\,0} + T_{\text{start}}) \\ &+ h_1(T_{\text{run}} + T_c + T_{\text{res}} + T_{\text{ndwait}\,1} + T_{\text{start}}) \\ &+ h_2(T_{\text{run}} + T_c + T_{\text{res}} + T_{\text{ndwait}\,2} + T_{\text{start}}) \end{aligned}$$

The results in Figure 7.26 compared to those of the tree composition in Figure 7.24 show that

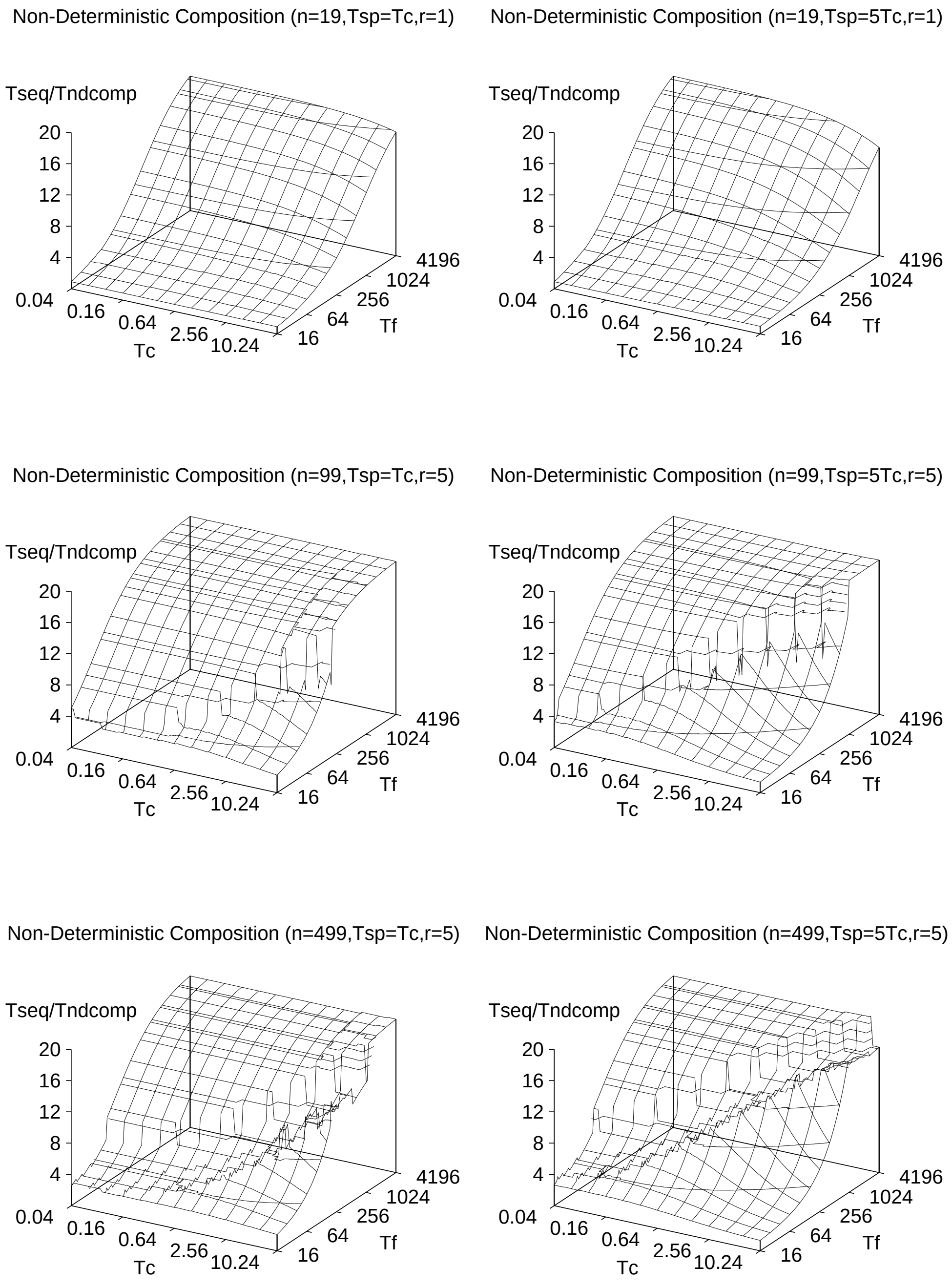


Figure 7.26: Non-Deterministic Composition Speedups

1. the speedup is significantly reduced for $n = 19$ and reaches only a level of 17 for $T_f = 4096$ (compared to 18 for tree compositions). The sequential execution path and the non-deterministic synchronization overhead are much too large in this case.
2. Only for very large n ($n = 499$) and corresponding high task granularities the hardware plateau is reached by exceeding a speedup of 19. For small T_f, however the speedup is still much worse.
3. T_c affects the speedup only for $T_f < 256$ ($T_{sp} = T_c$) or $T_f < 1024$ ($T_{sp} = 5T_c$). For very large T_f, this influence is not really significant any more.

In the diagrams for $n = 499$, a line of discontinuities indicates the switch from the approximation of the first case to that of the second case; the area between this line and the higher regions of the surface denotes a range where the speedup approximations are overly pessimistic.

7.7.8 Comparison

What are now the conclusions that we can draw from the analysis in this section?

1. **Iterated Task Composition.** It is the main task that starts all other tasks; consequently the processors are saturated in this sequential composition form earlier than in the corresponding parallel form. Likewise, in the combination phase, it is the main task that performs all synchronizations; no task blocking and awakening do occur which reduces the length of the critical path.

   On the other side, even if the combination phase has low synchronization overhead, this phase is still performed sequentially. Task creation rate does not adopt to the hardware resources, since the main task starts all other tasks in the initial phase. Since this main task occupies one processor, small modifications in the task granularity T_f and/or the task creation rate T_{sp} may cause surprising "jumps" in the total speedup.

2. **Parallel Composition.** Every task is started by its predecessor task. Therefore the task creation rate adopts to the number of processors; if all processors are busy, task creation is deferred. The speedup surface is smooth; small changes in T_f and/or T_{sp} can not cause sudden jumps.

   On the other side, it takes longer to saturate the system with work. The sequential combination phase has more synchronization overhead since each blocked task has to be awaked to perform the next combination step.

3. **Reverse Task Composition.** All tasks are started by the main task, therefore the system is saturated fast. Task creation overlaps with result combination; if the combination time is small enough, it may be cancelled out.

   However, for long combination times the speedup suddenly drops, because tasks are blocked and have to be awaked again like in parallel compositions. Like for iterated task compositions, the task creation rate does not adopt to the number of processors and the speedup figures may jump.

4. **Tree Composition.** All tasks are started by the main task; the system is saturated with work fast. The combination phase is parallelized which essentially reduces the length of the critical path. The programmer may control the task granularity.

   However, the number of tasks is twice as large as for the previous composition forms; the increased parallelization overhead reduces the speedup for fine-grained parallelism compared to reverse task compositions. Task creation rate does not adopt to the number of processors and speedup jumps may occur.

5. **Non-Deterministic Composition.** All tasks are started by the main task; the system is saturated with work fast. The combination phase is parallelized which essentially reduces the length of the sequential path. The programmer may control the task granularity.

   However, in addition to the larger number of tasks, it is the main task that performs all synchronization operations. This sequential bottleneck may considerably reduce the width of the combination tree, especially because the synchronizations are non-deterministic and have thus higher complexity. The task creation rate does not adopt to the number of processors and sudden speedup jumps may occur.

Above description is based on the simplifying assumptions made at the beginning of this section; constant T_{sp}, T_f and T_c and non-preemptive scheduling. We will now sketch the consequences if either of these constraints is released. Considering the complexity of the analysis even under these restrictions, it is not surprising that this sketch will be comparatively vague.

Non-Constant Execution Times

In general, the execution times of p, s, f and c are not constant but depend on the concrete arguments. If in a parallel composition form the maximum execution time of any of these functions differs very much from $1/n$-th of the sum of the execution times of all its applications, the analysis presented in this section is not really valid any more.

In *iterated task compositions and parallel compositions*, the value combination phase is sequential and begins when that task delivers a result that has been started last. In the constant time case, at this moment no other task is executing any more. In the general case at most $p - 1$ other tasks are still executing; only these tasks may delay the combination phase.

In *reverse compositions*, the sequential value combination phase begins with the first task that is started; any task that is executing longer than the average delays this phase. Most important, when the first task gets blocked by a delay of its predecessor task, its successor tasks get blocked as well; this makes the optimum non-blocking case unlikely.

In *tree compositions*, the execution of the main task and the creation of the combination tasks is mainly independent of variations in the task execution times. However, the height of the combination tree depends on the critical path that contains the tasks with the longest execution times for f and c. In the worst case, this path destroys the benefits of the parallelization of the combination phase.

Only *non-deterministic compositions* dynamically adapt to variations in the execution times by reordering the compositions such that tasks that terminate early are also composed early. The analysis in this section did not really do justice to this composition form that reveals its power only if some tasks possibly execute much longer than other tasks. In contrast to all other forms, non-deterministic compositions are only delayed when there is no more pair of task results available for combination.

Preemptive Scheduling

If tasks are preemptively descheduled at fixed time slices, the main task in iterated task compositions and in reverse, tree, and non-deterministic compositions is not any more favored above the subtasks it has generated:

First the task creation rate is effectively reduced. If n_t tasks are ready for execution at a certain moment, the main task gets only 1 time slice out of n_t to start new tasks; when the main task is scheduled again it only gets 1 time slice out of n_{t+1} slices to start the next tasks. Therefore, the more tasks are ready for execution, the fewer new tasks are started. If tasks terminate and the number of tasks gets low, the main task gets again more time for task creation.

Furthermore, since the main task has to compete for a processor like all other tasks, the speedup figures do not jump any more for small variations of the execution times; the speedup surfaces get much smoother.

In tree compositions and non-deterministic compositions, preemptive scheduling may let the combination tasks execute before necessarily the original tasks have terminated; value generation phase and value combination phase may thus proceed in parallel. This may become important for very long combination tasks that may so start execution earlier.

However, this may have also negative effects for non-deterministic compositions: when the main task is awaked after it has been blocked in a wait operation, the original tasks need not have terminated yet. Consequently, the main task may get blocked again and again which increases the total synchronization complexity and thus the parallelization overhead.

In general, preemptive scheduling increases runtime overhead because of the additional context switches that have to be performed.

7.8 Summary

Considering the many factors relevant, it should be clear that there are no simple rules able to tell under which conditions which form of parallel composition is actually best. However, at a guess

- if T_c is small and n is not very large, the simple composition forms with sequential combination phases should be preferred because of the lower parallelization overhead;
- if reverse composition is possible, it should be preferred above the other simple forms because of the overlapping of the task creation phase and of the combination phase;
- for large T_c or very large n, tree composition should be preferred over the simple forms because of the parallelization of the combination phase;
- non-deterministic composition should be preferred over tree composition only if there is evidence that the increased parallelization overhead pays off (i.e. only if the task execution times vary very much).

In general, the granularity of tasks (i.e. T_f) should be in the order of magnitude of 1000 to guarantee that the parallelization overhead does not limit the speedup. Granularities in the order of magnitude of 100 may still yield good speedups in certain cases. In above rules, T_c can be considered "small" if it is not larger than $T_f/100$.

Preemptive scheduling has the effect of smoothing the speedups by making them less dependent on (unfavorable but also favorable) scheduling orders. This is achieved at the price of increased scheduling overhead.

Chapter 8

Application Examples

In this section we demonstrate the parallelization of several functional programs. The first two examples (prime numbers and queens problem) are simple benchmarks; their efficient parallelization however raises several issues that are important for realistic programs as well. The remaining examples demonstrate the parallelization of several non-trivial computer algebra algorithms. The parallel programs were executed on a Sequent Symmetry S81 computer with 20 processors i386 running at 20 MHz and equipped with 128 MB of main memory.

8.1 Prime Number Computation

Figure 8.1 demonstrates an algorithm for prime number computation where a stream of primes is created in a cyclic fashion. The first element of this stream is 3, every remaining prime can be generated by resorting to previous stream elements only. The `para` annotation declares the recursively constructed stream `plist` and is the basis of the parallel algorithm: for every test number a task is started that decides about the primality of the number.

Unfortunately the compiler is not yet able to create the desired program. Since `plist` is declared as a task that returns a stream, by backward extension `sieve` and consequently `test` must return a stream. In `test` this would require the extension of the parameter `rest` to a task. The compiler generates this extension within `test` but issues a warning which indicates that the compiler has probably not created the intended parallel program.

The problem is that `rest` must be created as a task *before* `test` is called. The second `para` annotation in `sieve` ensures this interpretation; the compiler consequently makes a task from the recursive `sieve` call and constructs the non-strict parallel program. This difficulty shows the limitation of the backward extension algorithm which does not attempt to extend the actual task arguments.

The given program is a nice case study for the application of cyclic streams but is has no practical value since a task is created for every number to be tested. The basic idea may be more efficiently utilized by splitting the test range into *clusters* where one task tests all numbers in this cluster for primality. The result is a stream of clusters that are finally synchronized and concatenated.

This program is shown in Figure 8.2. The `test` function receives a cluster stream `clist` and a prime cluster `plist`. If `plist` does not suffice to decide about primality of the given test number, the next cluster is taken from the stream.

```
primes(N) where

  extern N :: Int -- maximum test element

  primes :: Int -> [Int]
  primes(n) =
    if n <= 3
      then [ ]
      else plist where
        para plist :: Task(Stream(Int)) -- ANNOTATION
        plist = 3 : sieve(n, 5, plist)

  sieve :: (Int, Int, [Int]) -> [Int]
  sieve (n, t, plist) =
    if t >= n
      then [ ]
      else test(t, plist, rest) where
        para rest :: Task(Stream(Int))  -- ANNOTATION
        rest = sieve(n, t+2, plist)

  test :: (Int, [Int], [Int]) -> [Int]
  test (t, plist, rest) =
    case plist of
      [ ]  => Error("test")
      p:pl =>
        if t < p*p
          then t : rest
          else if t % p == 0
            then rest
            else test(t, pl, rest)
```

Figure 8.1: Prime Numbers (Cyclic Variant)

```
append([], primes(N)) where

  extern N :: Int             -- maximum test element
  extern M :: Int             -- cluster size

  extern base :: Int -> [Int] -- generation of base primes
  B = base(M)

  primes :: Int -> [[Int]]
  primes(n) =
    if n < 3
      then [ ]
      else plist where
        para plist :: Task(Stream([Int]))    -- ANNOTATION
        plist = B : sieve(n, M, plist)

  sieve :: (Int, Int, [[Int]]) -> [[Int]]
  sieve (n, t, plist) =
    if t >= n
      then [ ]
      else cluster :@ rest where             -- ANNOTATION
        cluster = comp(test([ ], plist), [ ]) [ x | x <- t..t+M ]
        rest = sieve(n, t+M, plist)

  test :: ([Int], [[Int]], Int, [Int]) -> [Int]
  test (plist, clist, t, rest) =
    case plist of
      [ ]  =>
        case clist of
          [ ] => Error("test")
          c:crest => test(c, crest, t, rest)
      p:prest =>
        if t < p*p
          then t : rest
          else if t % p == 0
            then rest
            else test(prest, clist, t, rest)

  append :: ([*a], [[*a]]) -> [*a]
  append(l, r) =
    case l of
      [ ] => case r of
               [ ] => [ ]
               h:t => append(h, t)
      h:t => h:append(t, r)
```

Figure 8.2: Prime Numbers (Cluster Variant)

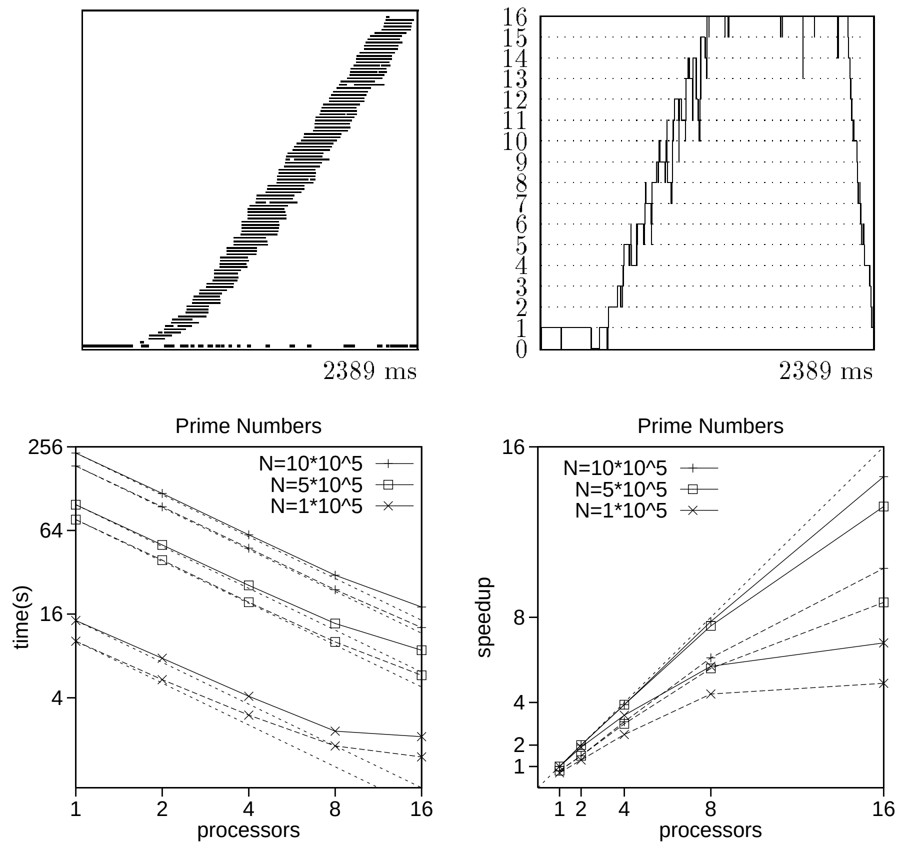


Figure 8.3: Prime Numbers

A subtle point in above program is the definition of `append` which allows the concatenation of the clusters to be performed *in parallel* with their concatenation. After the main program has started the task computing `plist`, it waits for the first cluster in the stream and traverses it to construct the initial portion of the result list. When the first cluster is exhausted, the task synchronizes with the next cluster and so on until the result list is computed.

It is tempting to define `append` in a more elegant fashion as

```
comp(app, [ ]) [ x | x <- l ]
```

where `app` concatenates two lists (this was the original approach of the author). However, the translation into a recursive function would then yield the body call

```
app(x, append(t))
```

with the consequence that the tail stream `t` is synchronized and concatenated *before* `x` is traversed and concatenated. Consequently, the concatenation would be performed in a final sequential step *after* `plist` has been delivered.

Unfortunately, the efficiency of the circular program is still not optimal compared to a sequential solution (which becomes visible during the benchmarks). The problem is that for every test test number at least one synchronization (for the first cluster of the stream) is

```
queens(N) where

  extern N :: Int    -- board size
  type Board = [Int]

  queens :: (Int) -> [Board]
  queens(n) = solve(n, 1, 1, [ ], [ ])

  solve :: (Int, Int, Int, Board, [Board]) -> [Board]
  solve (n, col, row, part, rest) =
    if col > n
      then rest
      else let
        sols = solve(n, col+1, row, part, rest)@ -- ANNOTATION
      in if conflict(col, 1, part)
        then sols!@                             -- ANNOTATION
        else if row == n
          then (col:part):sols
          else solve(n, 1, row+1, col:part, sols)

  conflict :: (Int, Int, Board) -> Bool
  conflict (col, dist, crest) =
    case crest of
      [ ] => False
      col0:rest =>
        if col == col0 || col == col0+dist || col == col0 - dist
          then True
          else conflict(col, dist+1, rest)
```

Figure 8.4: N Queens

required. It is actually better to pass the first cluster (with maximum prime P) directly to the tasks that may compute all primes (up to P^2) without any synchronization. Assuming $N \leq P^2$, the program can be rewritten in a non-circular fashion as:

```
primes :: Int -> [[Int]]
primes(n) =
  if n < 3
    then [ ]
    else B:@sieve(n, M, [B]) -- ANNOTATION
```

In the general solution, the algorithm would proceed in a sequence of iterations in each of which P_i tasks compute all primes in the interval $[P_i, P_i^2]$ and the results are synchronized by the main task.

These discussions already demonstrates a subtle issue when using a functional language for parallel programming: while various functions have the same denotation, they may have very different operational interpretations, in particular concerning the order in which expressions are evaluated or elements of a data structure are accessed.

While the compiler interprets the annotations given by the programmer to generate and synchronize tasks in an efficient way, it is entirely the responsibility of the programmer to provide a program whose operational interpretation and internal data dependencies allow parallel execution.

Figure 8.3 illustrates the runtime behavior of the two program versions vor various N and clusters of size $M = 1000$. The first two pictures visualize profiling information generated by the PACLIB kernel (for the circular version with $N = 100000$). They show the tasks active and the processors utilized during the program run, respectively.

The third picture displays the execution times (in seconds) for several values of N over a varying number of processors. We follow here the suggestion of [Crowl, 1994] by giving the absolute execution times and using logarithmic scales for both axes of the diagram. The times for the non-circular variant are denoted by the solid lines, those for the circular variant by the dashed lines. The dotted lines illustrate linear time scales. The last picture shows the speedups compared to sequential SACLIB C solutions.

The circular variant is always by a constant factor slower than the non-circular one. Both programs scale for large N almost linearly with the number of processors. The non-circular parallel program executed with 1 processor is virtually as fast as a corresponding SACLIB C program.

8.2 N Queens Problem

Figure 8.4 demonstrates the parallelization of the N Queens problem. The program is based on the doubly recursive function `solve` that takes a partial configuration `part` for the first `col` -1 columns of the board. It computes all compatible configurations where the queen in column `col` is placed in rows `row` to `n` and appends these configurations to the previously generated configurations `rest`.

The parallelization annotation `@` triggers the creation of a parallel process for the first recursive `solve` call. The synchronization annotation `@!` is used to synchronize the result of the corresponding conditional branch since since otherwise the other branches would be (superfluously) extended to task types as well.

However, the program is still extremely inefficient since it utilizes parallelism through all levels of the search tree. We solve this problem by redefining

```
queens :: (Int) -> [Board]
queens(n) = solve(n, 1, 1, [], [])

psolve :: (Int, Int, Int, Board, [Board]) -> [Board]
psolve(n, col, row, part, rest) =
  if row > P then
    app(solve(n, col, row, part, []), rest)
  else if col > n
    then rest
    else
      let sols = psolve(n, col+1, row, part, rest)@
      in if conflict(col, 1, part)
        then sols!@
        else if row == n
          then (col:part):sols
          else psolve(n, 1, row+1, col:part, sols)
```

where the `solve` function itself is not annotated. The key in this definition is that parallelism is only utilized in the first `P` levels of the search tree *and* that in the result expression

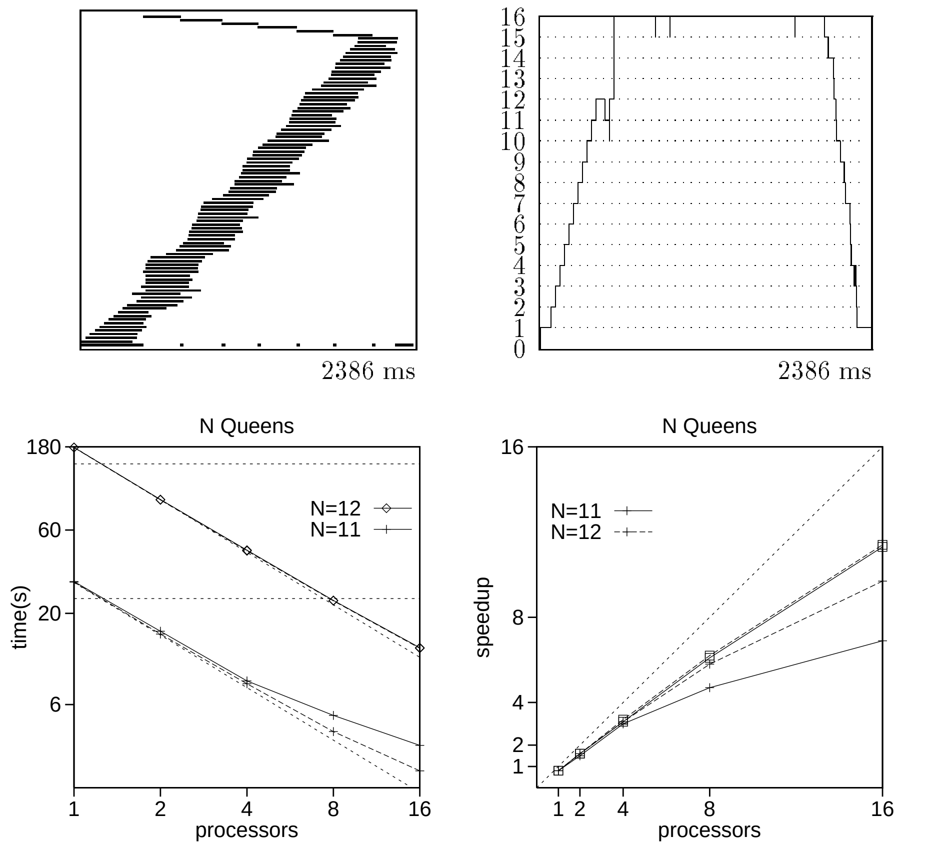


Figure 8.5: N Queens

```
app(solve(n, col, row, part, []), rest)
```

the synchronization of the task argument `rest` is *decoupled* from the application of the sequential `solve`. Consequently, first `solve` may be executed and then `app` may traverse the result list without synchronization. Only in the base level of recursion, `app` waits for the result of the `rest` task to perform the final concatenation and return the total result.

On the contrary, if this expression were defined as

```
solve(n, col, row, part, rest)
```

the `rest` task would be first passed through all applications of `solve` in the leftmost branch of the search tree. However, then `rest` must be synchronized to get the result list and traverse the rest of the search tree. Consequently, all tasks generated in the above `P` levels would essentially execute in sequence.

A minor problem with the para-functional program above is that the solution lists generated by the individual tasks are traversed multiple times to be concatenated in a binary-tree fashion. If we define instead

```
queens :: (Int) -> [Board]
queens(n) = append([], solve(n, 1, 1, [], []))
```

```
psolve :: (Int, Int, Int, Board, [[Board]]) -> [[Board]]
psolve(n, col, row, part, rest) =
  if row > P then
    solve(n, col, row, part, [])@:rest
  else if col > n
    then rest
    else
      let sols = psolve(n, col+1, row, part, rest)
      in if conflict(col, 1, part)
        then sols
        else if row == n
          then [col:part]:sols
          else psolve(n, 1, row+1, col:part, sols)
```

then `psolve` returns a lists of tasks each of which creates a solution list. The main task synchronizes with these results in an overlapping fashion (using the `append` function defined in the previous subsection) and traverses each solution list only once to concatenate them.

Figure 8.5 illustrates the runtime behavior of the two variants of the parallel program; the first two pictures refer to the second variant with sequential concatenation by the main task with $P = 2$ and $N = 11$ (i.e. about 11^2 tasks are generated).

The third diagram shows that both programs behave actually very similar except with smaller N and a large number of processors. In this case, the sequential concatenation variant (the dashed line) is clearly superior. In absolute terms, the parallel programs executed with one processor are about 25% slower than the optimal sequential program (illustrated by the horizontal dotted lines). This is mainly caused by the concatenation of result lists which are not required in the sequential solution. This is also the main reason why the speedup is sublinear although the parallel execution time scales very well.

8.3 Linear Equations Solver

Figures 8.6 and 8.7 demonstrate a parallel variant of the modular algorithm for the solution of a linear equation system; the sequential version was presented in Section 4.6 and Appendix C.1: Let A be a regular $n \times n$ matrix over $\mathbb{Z}$ and b a n vector over $\mathbb{Z}$. We want to find the unique n vector x over $\mathbb{Q}$ with $Ax = b$.

The basic idea is to apply Cramer's rule to transform the original problem into a problem of determinant computation. Determinants are computed by a modular algorithm that maps the problem from the infinite integer domain into several finite modular domains and computes the modular determinants. The Chinese remainder algorithm is applied to lift the modular determinants back into integer determinants from which the solutions of the equation system are computed. The algorithm is described in detail in [Stahl, 1992].

The parallel program proceeds as follows:

1. The main task computes a lower bound B for the product P of the primes p_i that have to be used to construct the modular domains.
2. The main task starts a set of `linDets` tasks (where `linDets` is an external C function used as a "black box"). Each task k projects the equation system (A, b) into the modular domain determined by p_k and computes $(d_k, \langle y_{k1} \dots y_{kn} \rangle)$. The prime p_k,

```
---------------------------------------------------------------------------
--
-- x = linSolve(A, b, n)
--
-- Solve a linear equation system ("parallel" version)
--
-- A: an Integer matrix of dimension n that is regular
-- b: an Integer vector of length n
-- n: a Digit
--
-- x: the unique Rational-Vector of length n such that Ax = b
--
---------------------------------------------------------------------------
linSolve(A, b, n, PRIME) where {

  -- program arguments (PRIME declared in saclib.i)
  extern A :: IntMatrix
  extern b :: IntVector
  extern n :: Digit

  -- libraries
  #include "dclib.i"
  #include "saclib.i"

  -- type names
  type IntVector = [Integer]
  type IntMatrix = [IntVector]
  type RatVector = [Rational]

  -- solve Ax=b for x
  linSolve :: (IntMatrix, IntVector, Digit, DigitList) -> RatVector
  linSolve(A, b, n, PLIST) = x where

    B = bound(A, b, n);

    base = ([], [], [], [], [])
    (dkyk, pk, Pk, ik, PREST) = comp(compose, base) [ (dy, p, P, PREST) |
        p:PREST <- PLIST, P = DTOI(1)
      ; ICOMP(P, B) == -1
      ; P' = IPROD(DTOI(p), P),
        dy = linDets(A, b, n, p)@ ]                        -- ANNOTATION

    d = cra_d(pk, ik, Pk, dkyk)@                           -- ANNOTATION
    ys = list [ cra_y(pk, ik, Pk, dkyk, i)@ | i <- 1..n ]@ -- ANNOTATION
    x0 = list [ RNRED(y, d)@ | y <- ys ]@                  -- ANNOTATION

    x = if comp(cand, True) [ d /= 0 | dy <- dkyk, (d,y) = dy ]
          then x0
          else linSolve(A, b, n, PREST)

}
```

Figure 8.6: Linear Equations (Part 1)

```
-- prime bound: P = 2*n!*m^n where m = max(Aij, bi)
bound :: (IntMatrix, IntVector, Digit) -> Integer
bound(A, b, n) = P where
  m0 = rcomp(IMAX, DTOI(0)) [ IABSF(x) | x <- b ]
  m1 = rcomp(IMAX, m0) [ IABSF(x) | l <- A | x <- l ]
  P = IPROD(DTOI(2), IPROD(IFACTL(n), IEXP(m1, n)))

-- modular computation of determinants
extern linDets :: (IntMatrix, IntVector, Digit, Digit) -> (Digit, DigitList)

-- auxiliary types
type DigitPair = (Digit, DigitList)
type DIL = ([DigitPair], DigitList, IntegerList, DigitList, DigitList)

-- composition of modular determinants, primes, products and inverses
compose :: ((DigitPair, Digit, Integer, DigitList), DIL) -> DIL
compose((dy, p, P, PREST), (dkyk, pk, Pk, ik, PREST0)) =
  (dy:dkyk, p:pk, P:Pk, i:ik, PREST1) where
    i = MDINV(p, MDHOM(p, P))
    PREST1 = case PREST0 of {[] => PREST; h:t => PREST0}

-- Chinese remainder algorithm on d component
cra_d :: (DigitList, DigitList, IntegerList, [DigitPair]) -> Integer
cra_d(ms, mis, Ms, as) = rcomp(cra2, DTOI(1)) [ (m, mi, M, d) |
    m <- ms, mi <- mis, M <- Ms, a <- as, (d,yi) = a
  ; d /= 0 ]

-- Chinese remainder algorithm on yi component
cra_y :: (DigitList, DigitList, IntegerList, [DigitPair], Int) -> Integer
cra_y(ms, mis, Ms, as, i) = rcomp(cra2, DTOI(1)) [ (m, mi, M, y) |
    m <- ms, mi <- mis, M <- Ms, a <- as, (d,yi) = a
  ; d /= 0
  ; y = nth(i, yi) ]

-- Chinese remainder algorithm on two arguments
cra2 :: ((Digit, Digit, Integer, Digit), Integer) -> Integer
cra2((m, mi, M, a), A) = MIDCRA(M, m, mi, A, a)
```

Figure 8.7: Linear Equations (Part 2)

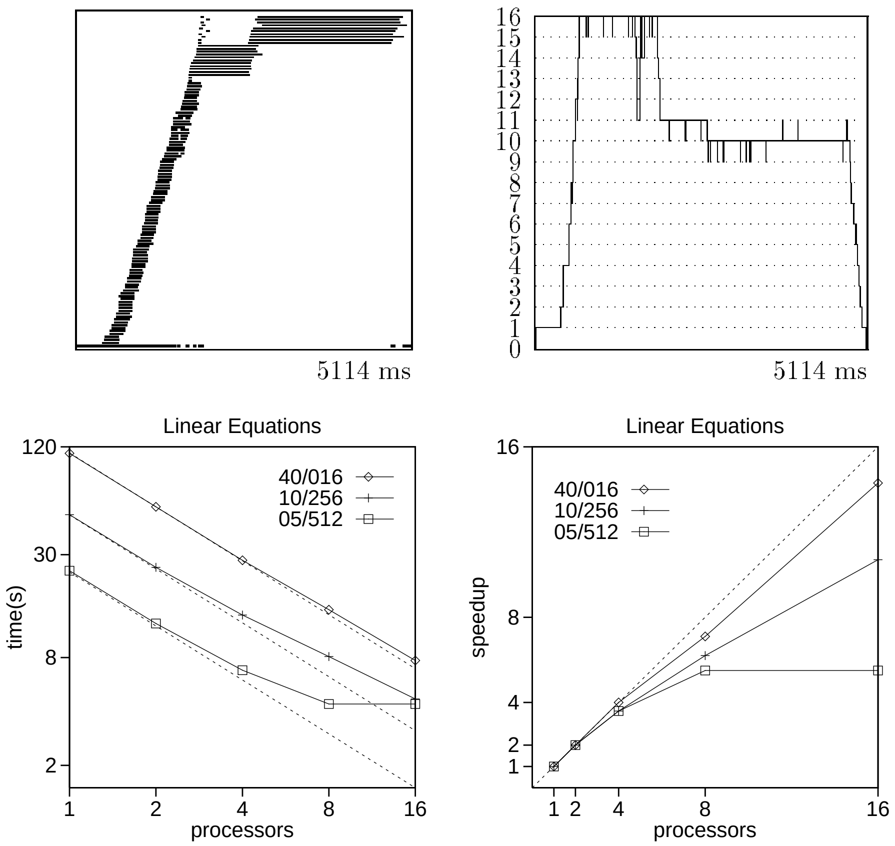


Figure 8.8: Linear Equations

the corresponding product P_k, and the inverse i_k are recorded as well as the start `PREST` of the unused part of the prime list.

3. The main task then starts a subtask `cra_d` (returning a value d), a task that starts n subtasks `cra_y` (returning n values y_i) and a task that starts n subtasks `RNRED` (that compute the actual result values $x_i = y_i/d$).

4. The main task finally checks whether all $d_i \neq 0$. If this test fails, the (very unlikely) case of an "unlucky prime" p_i has occurred. If this happens, the computation is repeated with new primes stored in `PREST`.

The outer annotations in the definitions of y_s and x_0 are not really required. They just ensure that $2n$ tasks are generated by two subtasks of the main task such that the main task can perform the check for unlucky primes as fast as possible.

Appendix C.2 displays the parallel target code generated by the compiler. Figure 8.8 illustrates the runtime behavior of the program for several values of D/L where D is the dimension of the equation system and L is the maximum bit length of the integer coefficients.

In the first picture (for $D = 10$ and $L = 256$), we can distinguish very well the three phases of determinant computation by `linDets` (the many short tasks), of Chinese remainder computation by `cra_y` and `cra_d` (the medium tasks) and of the reduction of the solutions to lowest terms by `RNRED` (the long tasks).

In the final stage, available parallelism is limited by D; therefore for low-dimensional systems the algorithm does not scale well for large number of processors. The execution times are in all cases virtually ($\pm 5\%$) the same as for the PACLIB C implementation of this method [Loidl, 1993].

8.4 Multivariate Polynomial Resultants

Let A and B be polynomials in $\mathbb{Z}[x_1, \ldots, x_{r-1}][x_r]$. The Sylvester matrix of A and B is a $(m+n) \times (m+n)$ matrix $M_{A,B}$ containing the coefficients $a_i, b_i \in \mathbb{Z}[x_1, \ldots, x_{r-1}]$ arranged in some special manner. The resultant of A and B is then $\det(M_{A,B}) \in \mathbb{Z}[x_1, \ldots, x_{r-1}]$.

[Hong and Loidl, 1994] implements in PACLIB several parallel algorithms for this problem; in the following we present the pD versions of the two most efficient algorithms that are parallelized on several levels.

The *first level* is the parallelization of the function `IPRES` (integer polynomial resultants). The core idea of both algorithms is to start a set of tasks `MPRES` (modular polynomial resultants) that map A and B into the homomorphic images $A_p, B_p \in \mathbb{Z}_p[x_1, \ldots, x_{r-1}][x_r]$ and compute $\det(A_p, B_p)$. The function `IPCRA` (integral polynomial Chinese remainder algorithm) then constructs $\det(A, B)$. Figure 8.9 shows the pD implementation of this idea by an annotated `rcomp` composition.

The two algorithms now differ in the parallelization of the lower levels.

8.4.1 Coefficient-wise Combine

The computation of $\det(A_p, B_p)$ in r variables proceeds recursively by computing in parallel k determinants (for some k) in $r-1$ variables and combining the sub-results by the function `MPINT` (modular polynomial interpolation). If r falls below some bound, a sequential algorithm is called. Figure 8.10 shows the pD function `pMPRES` that implements this idea.

However, since in both parallelization levels the combination phases `IPCRA` and `MPINT` are very time consuming, it is important to parallelize them as well. For both function, the same idea is applied: the coefficients of the (integral respectively modular) polynomial are processed in parallel. Figures 8.11 and 8.12 show the implementation of this idea by a simple `para` annotation that tells the compiler that a stream of values is to be generated.

8.4.2 Vandermonde Combine

As in the previous algorithm, the recursive structure of `MPRES` is parallelized by computing in parallel k determinants in $r-1$ variables. However, instead of combining the results by interpolation, a `stream` of these results is constructed (see Figure 8.13).

The external SACLIB C function `IVMMD` (inverse Vandermonde Matrix with modular digits) computes a $k \times k$ matrix that is constructed in a special way from the evaluation points used for the k determinants in $r-1$ variables. The rows and the matrix and the stream are forwarded to $k+1$ tasks that perform the necessary stream synchronizations to compute the vector products that determine the coefficients of the result polynomial.

```
------------------------------------------------------------------------------
--
-- Integral Multivariate Polynomial Resultant Computation
-- Based on PACLIB version by Hoon Hong
--
-- Arguments:
--   r:    a digit, r > 0
--   A, B: in Z[X1,...,Xr], A and B of positive degree in the main variable.
--
-- Result:
--  C: in Z[X1,...,X_{r-1}], the resultant of A and B w.r.t. the main variable.
--
------------------------------------------------------------------------------
pIPRES(r, A, B) where

  extern r :: Digit
  extern A :: IntPol
  extern B :: IntPol

  #include "saclib.i"

  ----------------------------------------------------------------------------
  -- integral polynomial resultant, parallel version
  ----------------------------------------------------------------------------
  pIPRES :: (Digit, IntPol, IntPol) -> IntPol
  pIPRES(r, A, B) = C where

    m = IPDEG(A)
    n = IPDEG(B)
    d = IEXP(IPMAXN(r, A), n)
    e = IEXP(IPMAXN(r, B), m)
    f = IDPR(IPROD(IFACTL(m+n), IPROD(d, e)), 2)

    C = rcomp(ICOMB, IntPolZero) [
          (MPRES(r, p, As, Bs), Q, p)
        |
          Q = IntOne, p:PREST <- PRIME
        ; ICOMP(Q, f) < 0
        ; Q' = if D then IPROD(Q, DTOI(p)) else Q,
          D = MPDEG(As) == m && MPDEG(Bs) == n,
          As = MPHOM(r, p, A), Bs = MPHOM(r, p, B)
        |@                                                     -- ANNOTATION
        ; D ]

    ICOMB :: ((ModPol, Integer, Digit), IntPol) -> IntPol
    ICOMB((Cs, Q, p), C) = C0 where
      q = MDINV(p, MDHOM(p, Q))
      C0 = IPCRA(Q, p, q, r-1, C, Cs)
```

Figure 8.9: Integer Polynomial Resultant Computation

```
-----------------------------------------------------------------------------
-- modular polynomial resultant, parallel version.
-----------------------------------------------------------------------------
pMPRES :: (Digit, Digit, ModPol, ModPol) -> ModPol
pMPRES(r, p, A, B) =

  if r == 1 then
    MUPRES(p, A, B)
  else if (r <= CUT) then
    MPRES(r, p, A, B)
  else C where

    m = MPDEG(A)
    n = MPDEG(B)
    m1 = MPDEGSV(r, A, 1)
    n1 = MPDEGSV(r, B, 1)
    k = m*n1+m1*n

    (C, D) =
      comp(MCOMB, (ModPolZero, ModPolOne)) [
        (pMPRES(r-1, p, As, Bs), i)
      |
        i = 0
      ; i <= k
      ; i' = if D then i+1 else i,
        D = MPDEG(As) == m && MPDEG(Bs) == n,
        As = MPEVAL(r, p, A, 1, i), Bs = MPEVAL(r, p, B, 1, i)
      |@                                                        -- ANNOTATION
      ; D ]

    MCOMB :: ((ModPol, Digit), (ModPol, ModPol)) -> (ModPol, ModPol)
    MCOMB ((Cs, i), (C, D)) = (C0, D0) where
      b = MDINV(p, MPTOD(MPEMV(1, p, D, i)))
      C0 = pMPINT(p, D, i, b, r-1, C, Cs)
      Dp = MPCONST4(1, 1, 0, p-i)
      D0 = MPPROD(1, p, D, Dp)
```

Figure 8.10: Modular Polynomial Resultants (Coefficient-Wise Parallelization)

```
--------------------------------------------------------------------------
-- integral polynomial Chinese remainder algorithm, parallel version.
--------------------------------------------------------------------------
pIPCRA :: (Integer, Digit, Digit, Digit, IntPol, ModPol) -> IntPol
pIPCRA(M, m, mp, r, A, a) =
  if r == 0
    then ITOIP(MIDCRA(M, m, mp, IPTOI(A), MPTOD(a)))
    else IPStdRep(R)
  where
    para R :: Stream(IntMon)                                -- ANNOTATION
    R = cra(IPMonoList(A), MPMonoList(a))
    cra :: (IntMonList, ModMonList) -> IntMonList
    cra(Ap, ap) =
      case Ap of
        [ ] =>
          case ap of
            [ ] => [ ]
            a:ap0 =>
              let (e, c) = a
              in (e, IPCRA(M, m, mp, r-1, IntPolZero, c)) : cra(Ap, ap0)
        A:Ap0 =>
          let (E, C) = A
          in case ap of
            [ ] => (E, IPCRA(M, m, mp, r-1, C, ModPolZero)) : cra(Ap0, ap)
            a:ap0 =>
              let (e, c) = a
              in if E > e then
                (E, IPCRA(M, m, mp, r-1, C, ModPolZero)) : cra(Ap0, ap)
              else if E < e then
                (e, IPCRA(M, m, mp, r-1, IntPolZero, c)) : cra(Ap, ap0)
              else
                (E, IPCRA(M, m, mp, r-1, C, c)) : cra(Ap0, ap0)
```

Figure 8.11: Integral Polynomial Chinese Remainder Algorithm

```
--------------------------------------------------------------------------
-- modular polynomial interpolation, coefficient-wise parallel version.
--------------------------------------------------------------------------
pMPINT :: (Digit, ModPol, Digit, Digit, Digit, ModPol, ModPol) -> ModPol
pMPINT(p, B, b, bp, r, A, a) =
  if MPDEG(B) == 0 then
    MPINV(r-1, a, 1)
  else if r == 1 then
    let d = MDDIF(p, MPTOD(a), MPTOD(MPEMV(1, p, A, b)))
    in if (d == 0)
      then A
      else let c = MDPROD(p, d, bp)
           in MPSUM(1, p, MPMDP(1, p, c, B), A)
  else
    MPStdRep(R)
  where
    para R :: Stream(ModMon)                                   -- ANNOTATION
    R = cra(MPMonoList(A), MPMonoList(a))
    cra :: (ModMonList, ModMonList) -> ModMonList
    cra(Ap, ap) =
      case Ap of
        [ ] =>
          case ap of
            [ ] => [ ]
            a:ap0 =>
              let (e, c) = a
              in (e, MPINT(p, B, b, bp, r-1, ModPolZero, c)) : cra(Ap, ap0)
        A:Ap0 =>
          let (E, C) = A
          in case ap of
            [ ] => (E, MPINT(p, B, b, bp, r-1, C, ModPolZero)) : cra(Ap0, ap)
            a:ap0 =>
              let (e, c) = a
              in if E > e then
                (E, MPINT(p, B, b, bp, r-1, C, ModPolZero)) : cra(Ap0, ap)
              else if E < e then
                (e, MPINT(p, B, b, bp, r-1, ModPolZero, c)) : cra(Ap, ap0)
              else
                (E, MPINT(p, B, b, bp, r-1, C, c)) : cra(Ap0, ap0)
```

Figure 8.12: Modular Polynomial Interpolation

```
---------------------------------------------------------------------------
-- modular polynomial resultant, Vandermonde parallelization
---------------------------------------------------------------------------
pMPRES :: (Digit, Digit, ModPol, ModPol) -> ModPol
pMPRES(r, p, A, B) =
  if r == 1 then
    MUPRES(p, A, B)
  else if (r <= CUT) then
    MPRES(r, p, A, B)
  else C where

    m = MPDEG(A)
    n = MPDEG(B)
    m1 = MPDEGSV(r, A, r-1)
    n1 = MPDEGSV(r, B, r-1)
    k = m*n1+m1*n

    Cb = stream [                                             -- ANNOTATION
           let As = MPEVAL(r, p, A, 1, i); Bs = MPEVAL(r, p, B, 1, i)
           in if MPDEG(As) == m && MPDEG(Bs) == n
             then pMPRES(r-1, p, As, Bs)
             else Error("Unlucky prime")
         |
           i <- 0..k ]

    -- inverse of Vandermonde matrix
    extern IVMMD :: (Digit, [Digit]) -> [[Digit]]
    P = IVMMD(p, list [ i | i <- 0..k ])                      -- ANNOTATION
    Cr = rlist [ (i, MDMPVP(r-2, p, Pi, Cb))@ | Pi <- P, i <- 0.. ]
    C = MPStdRep(Cr)

---------------------------------------------------------------------------
-- modular digit modular polynomial vector product
---------------------------------------------------------------------------
MDMPVP :: (Int, Digit, [Digit], [ModPol]) -> ModPol
MDMPVP(r, p, A, B) =
  if r == 0 then DTOMP(C) where
    C = rcomp(comb, 0) [ MDPROD(p, Ai, MPTOD(Bi)) | Ai <- A, Bi <- B ]
    comb :: (Digit, Digit) -> Digit
    comb (C0, C) = MDSUM(p, C, C0)
  else C where
    C = rcomp(comb, ModPolZero) [ MPMDP(r, p, Ai, Bi) | Ai <- A, Bi <- B ]
    comb :: (ModPol, ModPol) -> ModPol
    comb (C0, C) = MPSUM(r, p, C, C0)
```

Figure 8.13: Modular Polynomial Resultants (Vandermonde Parallelization)

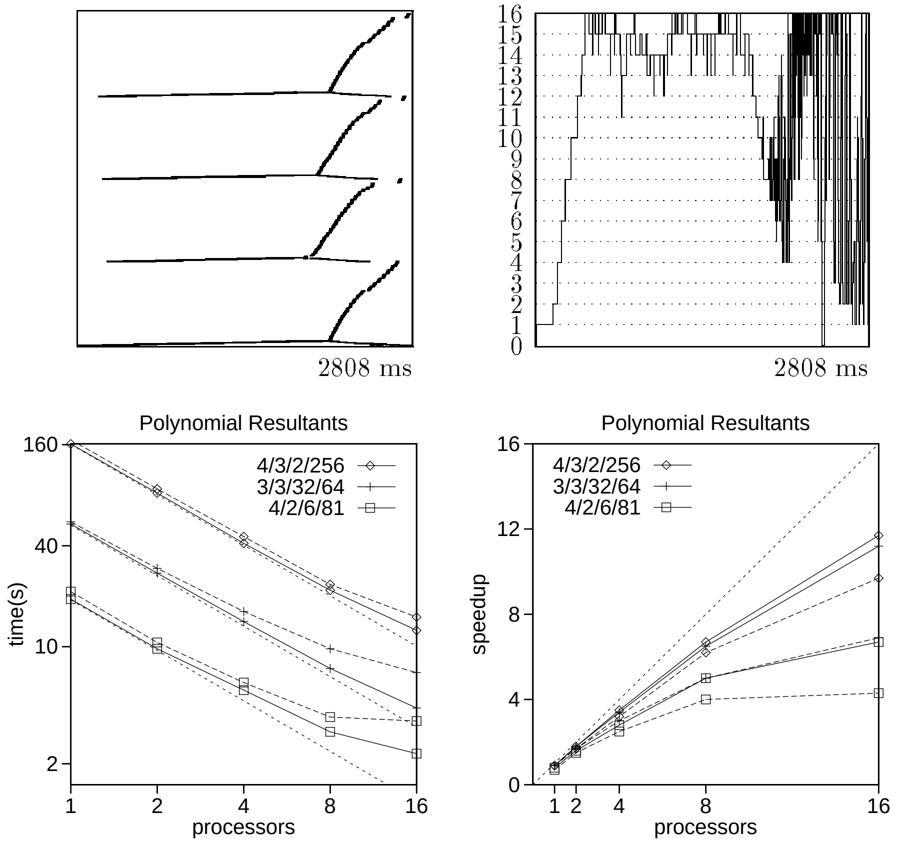


Figure 8.14: Polynomial Resultants

8.4.3 Results

The diagrams given in Figure 8.14 show almost the same execution times ($\pm 5\%$) for the functional programs as the C solutions described in [Hong and Loidl, 1994]. The keys in these diagrams describe the characteristics of the randomly generated polynomials A and B: 4/2/6/81 denotes polynomials with 81 monomials in 4 variables of maximum degree 2 with integer coefficient of at most 6 bits.

The sublinear speedups are a consequence of the parallelization overhead and of the algorithm which demands for minimum time complexity the sequential iteration of **MPINT** phases (where every iteration is parallelized as described above). Like in the C versions, the Vandermonde combine (denoted by the dashed lines) turns out to be less efficient than the coefficient-wise combine (denoted by the solid lines).

These factors that limit the parallelization do less depend on the language and its compilation but more on the algorithm; they affect the C implementation in the very same way. Actually we achieve in the coefficient-wise combine solution a slightly higher speedup than the C program by the application of virtual tasks that allow finer grained parallelism.

The size of the source code of the functional programs described in the section is about 1/3 of the corresponding PACLIB C source (without comments and without the main program that is in charge of input/output). The generated C target code is about twice the size of the manually coded C program; the same ratio applies to the corresponding object files. This increase in size is mainly a consequence of inlining.

8.5 Q-Matrix Computation

Let p be a prime and A be a monic univariate polynomial over $\mathbb{Z}_p[x]$ of degree n. Then the matrix $Q := (q_{ij})_{i=0}^{n-1}$ is uniquely defined by the following relation (for all $0 \leq i \leq n-1$)

$$\sum_{j=0}^{n-1} q_{ij}x^j \equiv x^{ip} \pmod A$$

i.e. row i of the matrix consists of the coefficients of the polynomial $x^{ip} (\text{mod } A)$. The Q matrix is required as an intermediate result in several computer algebra algorithms e.g. in Berlekamp's method for factorizing univariate polynomials.

In the sequential algorithm for this problem [Knuth, 1981], an auxiliary table of remainders $x^k (\text{mod } A)$ is constructed for $k = n \ldots 2n-2$. In the subsequent computations of polynomial products, powers of x are replaced by the corresponding remainders stored in the table:

Row 0 of the Q matrix is $x^0 = 1$. If $p \leq 2n-2$, row 1 is already determined by the auxiliary table. If $p > 2n-2$, row 1 is computed by iteratively squaring $x^{2n-2} (\text{mod } A)$ until the exponent of the result polynomial is as close to p as possible. This intermediate result is multiplied by x until $x^p (\text{mod } A)$ is available. All polynomial products are simplified using the auxiliary table.

Every subsequent row $x^{kp} (\text{mod} A)$ of the matrix is determined by multiplying its predecessor row $x^{(k-1)p} (\text{mod} A)$ with x using again the auxiliary table for simplifications.

[Minimair, 1992] describes a parallel PACLIB variant where the auxiliary table and the two first rows of the matrix are computed as in the sequential version. The parallelization idea is that, if rows i and j (i.e. the polynomials x^{ip} and x^{jp}) are available, row $i+j$ can be computed as the polynomial product $x^{(i+j)p} = x^{ip} * x^{jp}$.

A *static* distribution of rows among a fixed number of tasks yields poor performance since the time for the multiplication of polynomials essentially depends on the exponents. Therefore a scheme is developed where the main task maintains an auxiliary table that records which rows are available, which are under computation, and which remain to be computed. If a task returns row i as a result, this table is investigated for known rows j such that new tasks for unknown rows $k = i + j$ can be started.

The pD version of this algorithm starts a task for every row of the matrix. Rows 0 and 1 are computed as in the sequential algorithm. Task $k > 1$ computes its result by squaring the result of task $k/2$ (if k is even) or multiplying the results of tasks $\lfloor k/2 \rfloor$ and $\lfloor k/2 \rfloor + 1$ (if k is odd). Every task k depends therefore on at most two tasks i with $i < k$. Beginning with a single task, the number of tasks ready for execution therefore grows exponentially.

As illustrated in Figure 8.15, this idea is expressed by a a cyclic construction of the Q matrix `QM` (actually of the underlying polynomial sequence `QP`) which is similar to the parallel algorithm for prime number computation presented in Section 8.1. A single `para` annotation suffices for the compiler to extend every row to a task and to create the necessary task synchronization operations.

The results of the parallelization are described in Figure 8.17:

```
--------------------------------------------------------------------------------
--
-- Q-Matrix Computation
-- Based on SACLIB/PACLIB Version by Manfred Minimair
--
-- Arguments:
--   p: a digit, prime
--   A: in Z/p[X] of degree n
--
-- Result:
--   Q: a n*n matrix over Z/p such that, for all i, sum(Qij*X^j) = X^(ip) mod A
--
--------------------------------------------------------------------------------
qmatrix(p, A) where

  extern p :: Digit
  extern A :: ModPol

  #include "dclib.i"
  #include "saclib.i"

  qmatrix :: (Int, ModPol) -> [DigitList]
  qmatrix(p, A) = QM where

    n = MPDEG(A)
    X1 = MPCONST2(1, 1)
    Xn = MPCONST2(n, 1)

    (Q, R) = MPQR(1, p, Xn, A)              -- auxiliary table
    TM = rlist [ P
       | i <- 1..n-1,
         P = R ;; P' = R, (Q, R) = MPQR(1, p, MPUP(1, p, P, X1), A) ]

    Y = if p < n then                       -- first row x^p
          MPCONST2(p, 1)
        else if p <= 2*n-2 then
          nth(2*n-p-1, TM)
        else Y1 where
          k0 = ILOG2(IQ(DTOI(p), DTOI(2*n-2)))-1;
          Y0 = loop(head(TM)) [ Z
             | i <- 1..k0,
               Y = head(TM) ;; Y' = Z, Z = remsubst(MPEXP(1, p, Y, 2)) ]
          k1 = p-(2*n-2)*ITOD(IEXP(DTOI(2), k0))
          Y1 = loop(Y0) [ Z
             | i <- 1..k1,
               Y = Y0 ;; Y' = Z, Z = remsubst(MPUP(1, p, Y, X1)) ]

    para QP :: Task([Task(ModPol)])         -- Q matrix, ANNOTATION
    QP = ModPolOne : Y : list [ row(i, QP) | i <- 2..n-1 ]
    QM = list [ coeffs(P) | P <- QP ]
```

Figure 8.15: Q-Matrix Computation (Part 1)

```
-- polynomial coefficients
coeffs :: ModPol -> DigitList
coeffs(Y) = Q1 where
  k = n-MPDEG(Y)-1
  Q0 = list [ 0 | i <- 1..k ]
  Q1 = rcomp(cons, Q0) [ x | x <- MPCL(Y) ]

-- compute row i
row :: (Int, [ModPol]) -> ModPol
row(i, QP) =
  if i%2 == 0 then
    let Q = nth(i/2+1, QP)
    in remsubst(MPUP(1, p, Q, Q))
  else
    let Q0 = nth(i/2+1, QP)
        Q1 = nth(i/2+2, QP)
    in remsubst(MPUP(1, p, Q0, Q1))

-- remainder substitution
remsubst :: ModPol -> ModPol
remsubst(B) = MPSUM(1, p, S, R) where
  (S, R) = rcomp(sum, (MPCONST2(0,0), B)) [
           (MPUP(1, p, head(T0), MPCONST2(0, MPLDCF(R))), R0)
         |
           R = B, i = MPDEG(R),
           T = TM,
           N = n-2,
         ; i >= n ;
           R' = R0, R0 = MPRED(R),
           T' = T0, T0 = ntail(N-i+n, T),
           N' = i-n,
         ]

  sum :: ((ModPol, ModPol), (ModPol, ModPol)) -> (ModPol, ModPol)
  sum((A, R), (B, S)) = (MPSUM(1, p, A, B), R)
```

Figure 8.16: Q-Matrix Computation (Part 2)

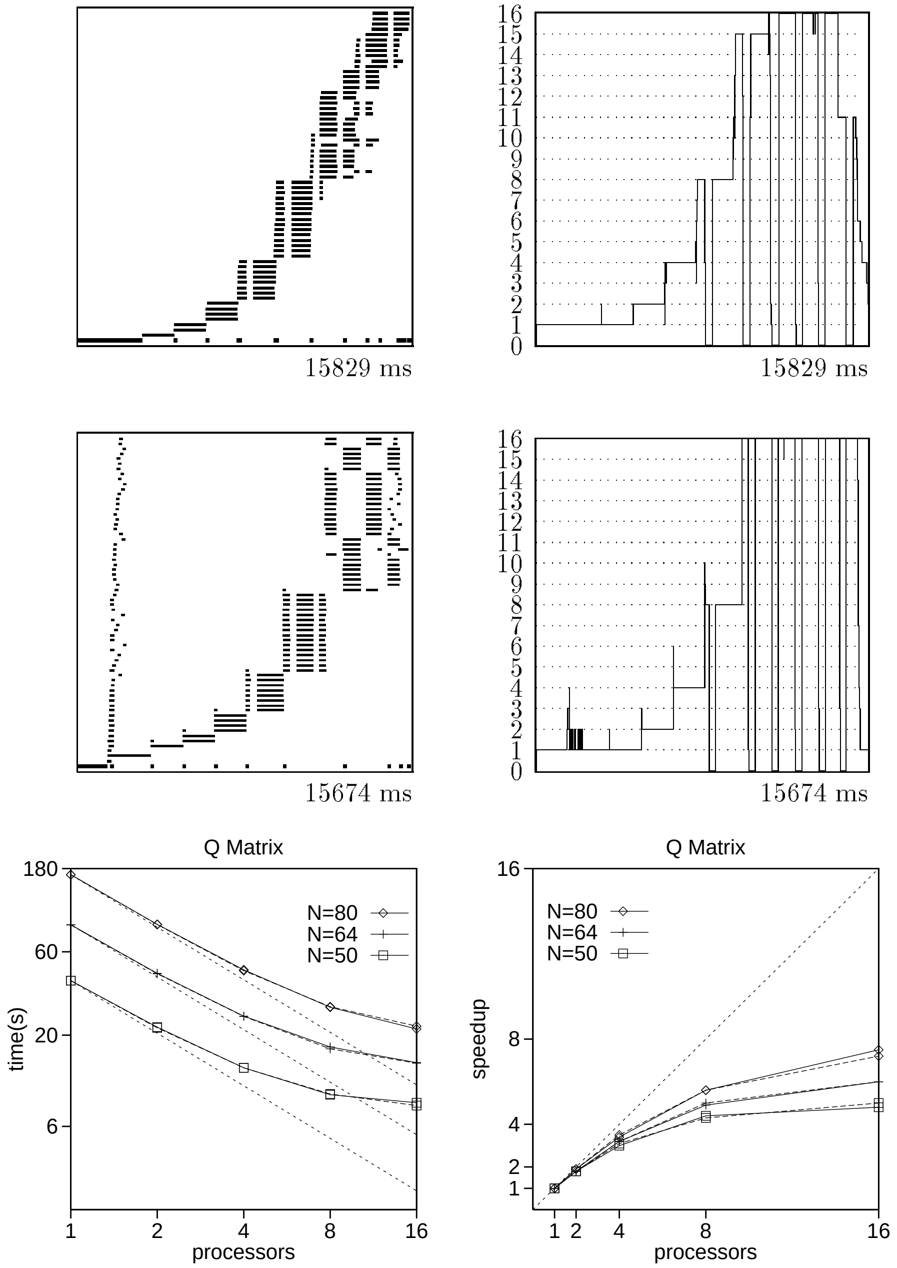


Figure 8.17: Q Matrix Computation

- The top diagrams illustrate the dynamic behavior and processor utilization for the original PACLIB solution for a random polynomial A with $n = 50$ and $p = 211$. The gaps with zero utilization show garbage collections.
- The medium diagrams give the same information for a run of the pD program on the same input arguments. In spite of the implementation differences, the dynamical behavior is almost the same.
- The bottom diagrams illustrate the execution times and speedups for both programs and various n (garbage collection times excluded). The solid lines denote the pD solution and the dashed lines the PACLIB program (both curves almost collide[1]). The dotted lines represent linear speedups.

Both programs show very similar behavior and give virtually the same performance. The speedup limit is caused by the startup phase where the auxiliary table is constructed and of the initial steps of the algorithm where only few tasks are active. It is therefore a consequence of the problem respectively algorithm and not of the parallelization overhead.

The pD source program is about half the size of the PACLIB C program; its only change compared to the D implementation of the sequential algorithm are the introduction of the function `row` and the definition of `QM` that originally was

```
QM = coeffs(ModPolOne) : coeffs(Y) :
   list [ coeffs(P) |
     i <- 2..n-1,
     P  = remsubst(MPUP(1, p, Y, Y)) ;; P' = remsubst(MPUP(1, p, P, Y))
   ]
```

On the contrary, the structure of the PACLIB program had to be fundamentally modified compared to the original SACLIB solution; the size of the source code increased by 40%.

[1] [Minimair, 1992] reports longer sequential execution times and thus higher parallel speedups. It seems that the unoptimized version of the SACLIB library has been used for comparison there.

Chapter 9

Related Work

In this chapter, we wish to give proper credit to related work of other researchers. The topic of this thesis is in the intersection of parallel computation, computer algebra, and functional programming. Consequently, our work has been subject to influences from all these areas for which there is currently only a single common forum, the *PASCO Symposium on Parallel Symbolic Computation* [Hong, 1994].

Section 9.2 is based on material from [Schreiner, 1993b].

9.1 Parallel Computer Algebra Systems

There are several international symposia that have served as a forum for research in parallel computer algebra: the *Computer Algebra and Parallelism* series [Della Dora and Fitch, 1988, Zippel, 1990] and the already mentioned *PASCO* symposium are especially dedicated to parallel algebraic and symbolic computation. The *DISCO Design and Implementation of Symbolic Computation Systems* series [Miola, 1993] presents work on both sequential and parallel computer algebra systems and also the *ISSAC International Symposium and Algebraic Computation* series [ISSAC, 1994].

While there are many ad-hoc approaches for implementing parallel computer algebra algorithms using available languages and tools, there are only a few coherent projects on designing parallel systems that especially meet the demands of computer algebra.

The L-project [Buchberger, 1978, Buchberger and Hintenaus, 1985] was one of the earlier approaches in this area. It pursued the design of a complete hard- and software system for parallel symbolic computation resulting in the design of a switch-based modular parallel architecture [Buchberger, 1985] and a parallel programming environment [Hintenaus, 1985]. The L-system was especially designed for the parallel implementation of recursion by a language that allowed the recursive description of networks of L-modules.

Several parallel systems have emerged from SAC-2 (the predecessor of SACLIB) that has been one of the the earliest packages for the implementation of algebraic algorithms [Collins and Loos, 19xx]:

PARSAC [Küchlin, 1990a, Küchlin, 1990b, Küchlin and Nevin, 1991] is a multi-threaded variant of SAC-2 originally based on the thread package of the operating system Mach; later it has been ported to the POSIX thread standard for Unix systems. PARSAC has implemented an automatic memory management scheme based on the concept of "preventive garbage collection" where heap memory is organized in pages; when a thread terminates, it copies its result into the page set of the parent thread and deallocates its own page set.

PARSAC has been in several respects a model for PACLIB; however, the PACLIB memory and task management is quite different and more general. Recently PARSAC has adopted the PACLIB garbage collection scheme [Bündgen *et al.*, 1994]. [Küchlin and Ward, 1992] have introduced (following the approach of [Mohr *et al.*, 1990]) the concept of virtual threads into PARSAC; PACLIB has adopted and generalized this method.

[Seitz, 1988, Seitz, 1990] presents a distributed version of SAC-2 for networks of workstations. Communication uses the SAC-2 input/output interface; parallel programming is based on concurrent procedure calls whose results are represented as futures [Halstead, Jr., 1985].

[Hong, 1991] has developed for the parallelization of the quantifier elimination algorithm a distributed version of SAC-2. The parallel programming model relies on a manager/worker scheme where the manager may interrupt the workers if their result is not required any more (speculative parallelism).

[Kredel, 1994] reports on a parallel implementation of SAC-2 for the KSR-1 virtual shared memory computer. The parallel programming model is based on the POSIX thread concept. To increase data locality and reduce communication latencies, heap structures are not shared between threads but copied into the local cache of the processor executing the thread. Garbage collection uses a mark and sweep scheme similar to the PACLIB kernel but confined to the local processor.

Sugarbush [Char, 1990, Char *et al.*, 1994, Burke-Perline, 1994] is a variant of the computer algebra system Maple parallelized with the programming model Linda [Carriero and Gelernter, 1990]. The Linda tuple space serves as the communication medium between concurrent Maple processes. Parallel programming is done in C; the sequential computer algebra code implemented in the Maple language and executed by the Maple interpreter. The system runs on shared memory machines and workstation clusters.

While most other approaches are based on concurrent variants of C, ||MAPLE|| (read "parallel Maple") [Siegl, 1992, Siegl, 1993b, Siegl, 1993a] has been parallelized with the help of the guarded Horn clause language Strand [Foster and Taylor, 1989]. The Strand interpreter cares for the communication between multiple Maple processes that execute the sequential portions of the algorithm. Parallel programming is done on the Strand layer by annotations of predicate calls. Strand supports non-strict streams, and non-determinism and is available on a variety of parallel architectures. Consequently ||MAPLE|| has been ported to various shared and distributed memory architectures and workstation clusters.

[Watt, 1986] presents a version of a multi-processor version of Maple that is able to distribute processes over a local area network. Parallel programming is based on an imperative message passing scheme.

MuPAD [Kemper *et al.*, 1991, Fuchssteiner, 1992, Fuchssteiner *et al.*, 1993] is a computer algebra system whose Maple-like language provides a "forall" construct for data parallelism. The system is implemented on shared memory.

[Willis and Bogucz, 1993] describes a coupling of the computer algebra system Mathematica with a nCube/2 distributed memory system; numerical C programs are executed in parallel, while Mathematica is mainly used for visualization.

PAC [Roch, 1988, Roch, 1990] is a computer algebra package running on several distributed memory systems. Special emphasis is given to the efficient implementation of arbitrary precision arithmetic by assembly routines and by utilizing available vector hardware. PAC++ [Gautier and Roch, 1994] is a further development of this system based on C++.

GENW2 [Tan and Wang, 1988] is a parallel code generator written in Lisp running under the Vaxima symbolic computation system. Given high-level algorithm specifications and symbolic expressions, GENW2 outputs code for execution on the Warp systolic array computer. The main target area of this system is finite element analysis. [Wang, 1991] presents GENCRAY, a version of this system for Cray vector-processors.

Carl Ponder [Ponder, 1988c, Ponder, 1988b, Ponder, 1988a] evaluates the (at that time) current status of parallel computer algebra. Ponder concludes that the existing parallel systems do not provide significant performance enhancements. Moreover, he argues that most efficient computer algebra algorithms are not suitable for parallelization.

The STURM library [Hong *et al.*, 1994a] is a C++ based successor of SACLIB for developing algorithms for algebraic constraint solving. Based on the PACLIB kernel, a multi-processor version of the STURM kernel has been developed [Hong *et al.*, 1994b]. A distributed version of this kernel for workstation clusters is currently under construction [Bertoli *et al.*, 1994].

9.2 Parallel Functional Programming Languages

Parallel programming in functional languages has a long research history covering a wide spectrum of areas: computer architecture (dataflow and graph reduction machines), abstract machines, runtime systems, garbage collection, language design, compiler construction, automatic parallelization, domain theory, strictness analysis, weight analysis, etc.

[Richards Jr., 1984] is a bibliography on functional programming that contains most of the early approaches of parallel implementations of functional languages. [Joy, 1990] is a more recent bibliography that contains some hundred citations on the parallel evaluation of functional programs. The annotated bibliography [Schreiner, 1993b] contains about 400 references on parallel functional programming, most of them with their full abstracts.

There are several symposia where both sequential and parallel implementations of functional languages are presented: In the conference series *FPCA Functional Programming and Computer Architecture* [FPCA, 1993] emphasis has shifted from special-purpose parallel architectures for functional languages to software implementations on conventional parallel machines. In the series *ACM Symposium on Lisp and Functional Programming* [ACM Lisp, 1992] sequential and parallel implementations of functional languages are discussed. Especially dedicated to parallel functional programming are the *International Workshop on the Parallel Implementation of Functional Languages* series [Workshop, 1993] and the new conference on *High Performance Functional Computing* [Feo and Oldehoeft, 1995].

Also conference series with a broader scope such as the *ESOP European Symposium on Programming* [Sannella, 1994], the *POPL Symposium on Principles of Programming Languages* [POPL, 1994], the *SIGPLAN Conference on Programming Language, Design and Implementation* [SIGPLAN, 1994] and the *SIGPLAN Symposium on Principles and Practice of Parallel Programming* [PPoPP, 1993] present related research results.

The fully automatic parallelization of functional languages is rather difficult. Consequently many language extensions have been suggested in order to give the programmer the possibility to be more explicit about his parallelization ideas and wishes, respectively. In this section, we will concentrate on this aspect of the spectrum that has been most influential to our work.

According to their effects on the functional subset of the language, we may classify the language extensions as follows:

1. **Explicit Parallel Constructs.** In functional languages with imperative extensions (like Lisp or ML) the well-known parallel extensions of sequential languages (e.g. message passing constructs) may be introduced. Parallel programming thus takes place on an imperative level but may profit from the functional language subset. We will not discuss this approach in this section.
2. **Functional Parallel Constructs.** Even in not purely functional languages, it may be the case that all parallel constructs assume that the involved functions do not cause any side-effects. Provided the correct application of these constructs (that the programmer is in charge of), this yields an essentially functional programming style (see Section 9.2.1).
3. **Data Parallel Constructs.** On SIMD (Single Instruction Multiple Data) architectures, these parallel language extensions usually have to be restricted to certain forms of data parallelism where the same operation is applied to all elements of an aggregate (e.g. array) (see Section 9.2.2).
4. **Parallel Annotations.** If the basic language is purely functional, parallel language extensions turn into mere program annotations that do not influence the correctness of a result but only the efficiency of its computation. The programmer may insert or remove such annotations without any risk (see Section 9.2.3).

The last category is also called "Para-functional Programming" in a narrow sense [Hudak, 1986b]; however, the last three categories actually lead to a similar programming style with common characteristics.

Algorithmic skeletons are a related term coined by [Cole, 1989]. It denotes a set of predefined higher order functions that represent the structure of typical parallel algorithms. These algorithmic skeletons (that can be parameterized by the concrete application functions) may be efficiently parallelized and mapped onto the actual architecture (see Section 9.2.4).

A related category are **single-assignment languages** that have evolved from imperative languages by restricting the assignment operation and thus yield functional characteristics. Modern languages of this type are actually equivalent to functional languages; the only (mainly syntactic) difference is their explicit support of iterated loops (see Section 9.2.5).

Another issue is the introduction of **non-determinism** into functional languages to increase the parallelism in a program (see Section 9.2.6).

9.2.1 Functional Parallel Constructs

[Halstead, Jr., 1985, Halstead, Jr., 1986] introduces the Scheme dialect *MultiLisp* that provides *pcalls* for parallel argument evaluation and *futures* for concurrency between the computation of a value and its use. *future* starts a process evaluating an expression but immediately returns a reference to the result which is indistinguishable from the actual result. Any strict operator checks whether its argument is a future or not and blocks when the corresponding process has not yet terminated. MultiLisp was implemented by a byte-code interpreter on the Concert multiprocessor built at MIT; several variants were developed later, e.g. *MultiScheme* [Miller, 1987] and *Mul-T* [Kranz *et al.*, 1989]. In this context, [Mohr *et al.*, 1990, Mohr *et al.*, 1991] develop the technique of task inlining generalized to *lazy task creation*.

[Goldman and Gabriel, 1988, Goldman and Gabriel, 1989] describe the parallel Lisp dialect *Qlisp*: The *spawn* construct for process creation returns a future, but in contrast to MultiLisp the application of a synchronization construct is required to get the actual value of a future. The *qlet* construct evaluates several variable bindings in parallel. The evaluation of the body expression may optionally proceed concurrently with the evaluation of the bindings, but then the programmer has to explicitly apply the synchronization construct.

The notion of futures is also generalized in two ways: First, the results of several processes may be associated to a single future; as each process finishes, it calls a function that combines its result with the previous value of the future. Second, speculative parallelism is supported by providing an end-test predicate to a combining future. When the value computed by a process satisfies this test, all not yet finished processes associated to the future are killed. Section 7.6.5 proposes a PACLIB kernel extension based on this idea.

The *qlambda* construct is used to create a closure for a process. This closure may be used by several other processes to transmit *sequences* of values to the process and to receive *sequences* of results. It thus simulates a form of stream-based communication.

spawn, *qlet*, and *qlambda* have an extra argument signalling whether new processes shall be actually spawned or evaluations shall be performed locally. This can be used for controlling the amount of parallelism in divide and conquer programs. For the same purpose, there is a predicate saying whether the process queue is empty. [Pehoushek and Weening, 1988] describes techniques for low-cost process creation and dynamic partitioning of Qlisp programs.

9.2.2 Data Parallel Constructs

[Steele Jr. and Hillis, 1986] describes *Connection Machine Lisp*, a data-parallel SIMD dialect of Lisp. A datatype *xapping* is introduced which is an unordered set of index/value pairs. There are several operations to create, combine, and manipulate xappings. An FP-like "apply-to-all" operator applies a function to all elements of the xapping in parallel. A "fold" operator combines the elements of a xapping in a non-deterministic order into one.

[Merrall, 1991b, Merrall, 1991a] base an SIMD implementation of EuLisp on the data structure *paralation* [Sabot, 1988] which is a virtual processor set. Instances of this paralation are fields where each element is stored on a separate processor and where to each element the index of the corresponding processor is associated. Operations on these fields can be performed in parallel by each processor.

[Jouret, 1991] extends a functional language by an aggregate type and a set of primitive monolithic operations on this type. Communication and computation are clearly separated (by a higher-order operation *imap* and several shuffle operations) such that transformation rules can be easily applied.

[Hill, 1993] defines a data-parallel extension *DPHaskell* of the lazy functional language Haskell. A parallel data type POD (parallel object with arbitrary dimension) is introduced which represents an infinite set of index/value pairs. Parallel algorithms are expressed by POD comprehension where elements are drawn from generators lazily (i.e. only when required) to avoid a combinatorial explosion of parallelism.

[Blelloch, 1993, Blelloch *et al.*, 1993] describe *NESL*, a language that in contrast to the other data-parallel approaches also allows nested data-structures and *nested parallelism*. The language also provides mechanisms for calculating the asymptotic runtime of the program on various parallel machine models e.g. the parallel random access machine.

9.2.3 Parallel Annotations

[Burton, 1984] introduces three *application annotations* $@_E$, $@_L$ and $@_P$ for the eager, lazy, and parallel application of a (curried) function to its argument, respectively. These annotations give the programmer complete control over the reduction order of a functional program. [Burton, 1987] extends this set of annotations by explicit processor specifications and non-determinism: the *amb* operator non-deterministically returns one of its arguments as soon as this argument becomes available.

[Hudak, 1986b, Hudak and Smith, 1986] coin the term *para-functional programming* for functional programming with (semantics-preserving) parallelization annotations exemplified by the language ParAlfl. *Mapped expressions* are annotated with the identifier of the processor where evaluation takes place. *Eager expressions* are used to explicitly override the lazy evaluation strategy which is default.

[Hudak and Anderson, 1987] applies partially order multisets (pomsets) to define the operational semantics of parallel functional programs under different evaluation strategies. Based on this model an annotation language is sketched that allows the programmer to precisely control the evaluation of the program. [Hudak, 1991] refines this annotation language: each expression can be annotated by a *schedule* based on a partial order of *events* (demand of a value, start of the evaluation, end of the evaluation).

[Kewley and Glynn, 1989] describes *strictness annotations* for the parallel implementation of Hope+. The programmer may explicitly specify the degree of evaluation of an argument and also enforce the evaluation of an argument concurrently with the function body.

[Kelly, 1989] describes an annotation language where the programmer may explicitly specify the partitioning of a functional program into a set of processes. Only static process networks are considered whose parameters are fixed at compile-time.

[Maranget, 1991] describes a parallel implementation *GAML* of Lazy ML for shared memory multiprocessors. Parallelism is introduced by the programmer with the help of fork annotations.

[Nöcker *et al.*, 1991] defines the language *Concurrent Clean* that uses simple strictness annotations and parallelization annotations.

9.2.4 Algorithmic Skeletons

[Cole, 1988, Cole, 1989] propose the use of *skeletons* i.e. higher-order functions designed to capture typical parallel computation patterns, e.g. divide-and-conquer. Parallel programs are created by combining instances of these skeletons for which efficient parallel code can be generated.

[Darlington *et al.*, 1991, Darlington *et al.*, 1993] describe a programming methodology based on skeletons and program-transformation techniques. General functional programs are transformed into compositions of known skeletons and programs are converted from one skeleton to another, adapting it to the characteristics of the underlying architecture.

[Bratvold, 1992] discusses how to automatically identify algorithmic skeletons in functional programs and to extract the useful parallelism. [Bratvold, 1993, Michaelson *et al.*, 1993] present the *SkelML* compiler that exploits parallelism in certain higher order functions and in function composition and generates parallel Occam2 code. The PUFF system (Parallelization Using Farmed Functions) [Busvine, 1993] uses profiling information to identify opportunities for employing the processor farm skeleton. [Bratvold, 1994] demonstrates the parallelization of functional programs using a list-homomorphism skeleton.

[Rabhi, 1993b, Rabhi, 1993a] propagate the use of algorithmic skeletons as the "paradigm-oriented" approach to the design of parallel programs. They concentrate on process networks and static iterative transformation algorithms.

[Danelutto *et al.*, 1994] pursues the skeleton approach for a portable parallel programming environment.

9.2.5 Single Assignment Languages

[Ashcroft and Wadge, 1977, Wadge and Ashcroft, 1985] describe the single assignment language *Lucid*. The order of statements in Lucid programs is irrelevant, in fact, the language can be seen as a first-order functional language with a monomorphic type system. Special loops provide a syntactic sugar for tail-recursion. The language has been implemented on several multiprocessors.

[Chen, 1985, Chen *et al.*, 1991] describe the dataflow language *Crystal*. Crystal is a purely functional language that provides the special data types *index domains* which embody the geometric shape of data structures and *data fields* which generalize the notion of distributed data structures, unifying the notion of arrays and functions. The language has been implemented on several multiprocessors including hypercube systems.

[Szymanski, 1987, Szymanski, 1991] describe the "Equational Programming Language" EPL that is targeted for scientific applications. EPL programs are recurrent equations that define the elements of multidimensional arrays (or more general structures). Equations can be annotated with name of virtual processors indexed by the equation subscripts to assist the compiler in program parallelization. Code is generated for several SIMD architectures, shared memory machines and message passing computers.

[Oldehoeft and Cann, 1988, Feo *et al.*, 1990, Cann, 1992] describe the single assignment language *SISAL* (Streams and Iterations in a Single Assignment Language). SISAL can be seen as a strict first-order functional language with a monomorphic types system. The language provides loops and arrays which are the most important source of parallelism. An explicit stream data type is used for expressing pipelined parallelism. The language is explicitly targeted for scientific computation and has been implemented on data flow computers, shared memory multiprocessors, vector-supercomputers and SIMD machines.

[Arvind and Nikhil, 1987, Nikhil, 1991] describe the dataflow language *Id*. Id is a higher-order polymorphic functional language with a non-strict but non-lazy (i.e. *lenient*) evaluation semantics i.e. (conceptionally) each expression is evaluated by a concurrent process. The language includes *I-structures* [Arvind *et al.*, 1986] i.e. single assignment arrays on which other non-strict data structures (e.g. streams) can be built. Loops can be used for processing these arrays, but the language also provides high-level array comprehension. Id is the standard language of the MIT dataflow project and has been implemented to several dataflow machines.

Currently, efforts are going one to merge Haskell and the Id approach of parallelism to a new language *pH* [Nikhil *et al.*, 1993].

9.2.6 Non-Determinism

Basically there are two motivations to introduce non-determinism into a functional language:

1. **Internal Non-Determinism.** To increase parallelism by loosening the order of evaluation steps.
2. **External Non-Determinism.** To let functional programs deal with external events that asynchronously arrive.

The second issue arises in the application of functional languages for interactive systems (e.g. for implementing input/output or even functional operating systems [Turner, 1987]). However, we will concentrate on the research that relates to the first motivation:

Already [McCarthy, 1963] introduces the binary *amb* operator that non-deterministically returns one of its arguments and whose result is only undefined if both arguments are undefined.

[Henderson, 1980] uses the *or* operator (another name for *amb*) and the constant *none*. *or* chooses its return value in a "magical" way such that (if possible) the evaluation of *none* constants is avoided. With this feature backtracking algorithms can be easily implemented.

[Friedman and Wise, 1980] defines the *frons* operator that adds an element to a list at a position determined by the time its value becomes available. [Henderson, 1982] suggests a *merge* operator that non-deterministically interleaves two streams in the order in which their elements become available.

However, all these operators destroy the referential transparency of a functional program and the formal semantics becomes complex. The parallel or operator *por* that returns true when the first argument has evaluated to true and its counterpart *pand* do not suffer from this deficiency but their usefulness is limited [Søndergaard and Sestoft, 1988].

[Hudak, 1986a] proposes a new solution to the aggregate update problem involving non-determinism and concurrency. The construct *choose-one* is introduced that returns non-deterministically an element of a list and the list with the chosen element removed. Based on this primitive, the construct *apply-all* applies a list of functions in a non-deterministic order to an argument.

[Burton, 1989, Burton, 1991] describes an approach to handle non-determinism by a special type *improving* and a speculative *max* function that starts a parallel process for its second argument and returns its first argument as long as the the second argument is not available. After the process has terminated, the maximum of both arguments is returned. Using this primitive, algorithms with determinate result but indeterminate behavior can be implemented.

[Hughes and O'Donnell, 1989] introduces a data type *set* whose elements are the possible values of a non-deterministic expression. However, the set is represented only by one of its elements (whose evaluation has terminated first). Since the only operations allowed on this set are the singleton former, union, Cartesian product, and the map operation, equational reasoning about the program is still possible. However, this approach hides the problem more than actually solving it. At last at the termination of the program, the element of the set has to be selected and a non-deterministic choice is required.

[Clarke, 1990] investigates two classes of associative commutative operations which may be used to exploit the concurrency derived from indeterminate behavior: *simple aggregates* encapsulate the evaluation order indeterminacy implicit in strict associative commutative operations and have no effect on semantics. *Early Readable Aggregates* are aggregates from which information can be read before the evaluation is completed. This reading is restricted in order not to yield non-deterministic program results.

[Marino and Succi, 1989] proposes several data structures that are better suitable for parallel programs than the sequential list structures: *bags* (i.e. unordered collection of objects) with a non-deterministic access function, a folding operator and a mapping operator and *limited functions* with finite enumerable bags as domains. It is shown that programs over bags contain more inherent parallelism than programs over lists.

[Jayaraman and Plaisted, 1987] proposes the language SEL based on equations and sets for functional programming. The user is in charge of establishing a deterministic program semantics by giving annotations whether deterministic or non-deterministic rewriting is to be applied to certain rules.

[Barth *et al.*, 1991] suggests *M-Structures* to express non-deterministic behavior in a non-strict functional language. Each element of an M-structure is tagged with a state bit indicating whether the element is empty or full. Write operations set this bit, read operations unset it, read operations on an empty element are delayed. Using this implicit form of synchronization atomic modification operations can be defined and non-deterministic computations can be expressed. Barriers are used to localize the effects of non-deterministic computations to a binding block. It is shown that this approach yields in several cases to programs that are more efficient, more parallel and more declarative by not overspecifying the execution order.

[Banatre and Le Metayer, 1989, Banatre and Le Metayer, 1990] introduce the Γ (GAMMA) language which is based on the non-deterministic manipulation of multisets (bags). The language uses the model of a *chemical solution* in which floating molecules can interact with each other according to reaction rules. Technically, a program is a set of rules that are applied to the element of the multi-set in a non-deterministic way. Any tuple of elements that fulfills the prerequisites of a rule may be replaced by an element as specified by the rule. The program terminates when no more rule can be applied.

[Kuchen and Gladitz, 1992, Gladitz and Kuchen, 1994] describe the efficient distributed implementation of bags for several forms of Γ operations.

Chapter 10

Conclusions and Future Work

We have developed a para-functional programming interface that has been useful for parallelizing a few non-trivial computer algebra algorithms. The results show that the performance of these programs is not worse than the performance of programs that are coded with much more effort directly in a parallel dialect of C. We believe that in an application area where most algorithms are essentially functional (like in computer algebra) parallel programming with the pD system is feasible and yields programs that are more elegant and more amenable to experiments with different parallelization strategies. The development of further parallel computer algebra algorithms will show whether this claim is justified.

The pD programming model is original by its combination of several known approaches, their integration into a common framework, and own extensions. The programmer may write programs on several levels of abstractions:

1. On the *imperative* level in the PACLIB C language. This gives most flexibility but also requires tedious coding details. This level also serves as the compilation target; the compiler generates a monomorphically typed parallel program where it is known at any point whether an object is a task or not.
2. On the *pseudo-functional* level, using the `Task` type and operations for task creation and synchronization. Parallel programming on this level is much the same as on the imperative level but the high-level functional features of the functional core language (especially the iterated composition forms) may be utilized.
3. On the *para-functional* level using expression or type annotations without explicit `Task` objects. The compiler introduces the necessary `start` calls and retypes the program aggressively deferring synchronizations as far as possible (if necessary by generating duplicate functions with new type interfaces).
4. On the level of *iterated compositions* that are often the first targets of parallelization. The programmer may annotate composition forms to indicate the parallel traversal of iteration ranges. She may also use new forms where the order of compositions is less restrictive (tree composition or non-deterministic composition).

Expression annotations are similar to the application of futures, but the compiler transforms the annotated source program into target code where concurrent tasks are identified as a separate data type with explicit conversion (task creation and synchronization) operations; no further runtime checks are required. Using *parallel compositions* are similar to the application of algorithmic skeletons, but the constructs are less restrictive by overlapping the generation of variable bindings, the computation of values, and the combination of intermediate results.

The application of *type annotations* to express parallel programs is completely original to the pD system and often allows extremely elegant parallel programs by avoiding expression annotations scattered over the program. They are an offspring of the compiler's treatment of expression annotations as type disorders that have to be resolved. The compiler restricts parallelism not to the body of a function but allows tasks to migrate beyond the boundaries of functions and even other tasks. Synchronizations are created only in the last moment (at strict basis operations) or to avoid repeated synchronization of the same task.

The main problem with this style of programming is that there is no obvious relationship between the denotation of a program (*what* is computed) and its operation in the parallel context (*how* it is computed). Minor differences in the formulation of an algorithm may have drastic consequences on its synchronization behavior and thus on the parallelization. The same is true for different forms of para-functional annotations. Basically the programmer has to know which transformations and optimizations the compiler applies (and, perhaps more important, which not) to anticipate the dynamical behavior of a para-functional program. Without this knowledge, surprises will occur and frustration may take place.

This is an important point of compilation: It does not suffice that the compiler performs sophisticated transformations, but they must be also *intelligible* to the programmer to estimate which annotations suffice to achieve the desired effect. Para-functional programming is still parallel programming even and because it operates on a higher level of abstraction. This implies that the programmer must be able to rely on a simple model of what the compiler does. Such a model in pD is that, if a computation is extended to a task, this task is threaded through all functions and data-structures until its result is actually required. However, the relation between denotation and computation is an open area with much room for further improvements.

The work described in this paper does not deal with automatic parallelization. In fact, the more we have learned about the development of parallel functional programs, the more sceptical we have become on the possibilities of automatic parallelization in the application area under consideration. In our experience, virtually none of the functional formulations of a sequential computer algebra algorithm is directly amenable for parallelization. Almost always an algorithmic key transformation is required that is based on the underlying mathematical theory.

This transformation may be only a few lines in the source code, but they require an insight into the particular problem which goes beyond the capabilities of an automatic tool. This transformation is the actual parallelization and can be only performed by a programmer that already has a clear understanding of the parallel algorithm. The key contribution of this thesis is that, once this transformation has been made, the programmer needs to communicate the parallelization idea only by program annotations to the compiler which then cares for the tedious details.

On the one hand, we do not believe that an automatic tool (unless for other application aras which much more restricted algorithms) can perform the necessary key transformation. On the other hand, we do not see much point in applying an automatic parallelizer *after* the key transformation has been performed; the programmer may then easily provide the few additional annotations.

Our future work will concentrate on the further development of the system in several directions:

The pD language is currently based on lists as the predominant data structure. This does not any more reflect modern trends in computer algebra systems that also support and actually heavily rely on arrays. In fact, the new STURM kernel developed at our institute is based on a variant of the PACLIB kernel using arrays instead of lists. This requires the extension of the iterated composition forms and their parallel variants to arrays (similar to the *array comprehensions* of Haskell).

The pD system will then also support the STURM kernel as the underlying runtime system. To make the external interface of pD compatible with STURM, C++ like *overloading* will be supported. When the distributed version of the STURM kernel is available, the necessary language extensions will be introduced into pD to make it suitable for programming on workstation clusters. The kernel extension proposed in Section 7.6.5 will be introduced to make non-deterministic compositions much more efficient.

The parallel compositions will be extended to new forms and refined to give consistent granularity control for all variants (allowing a task to process multiple values). The parallel annotations will be extended to allow non-deterministic but task-transparent synchronization (i.e. we will support para-functional version of the `any` operator).

We will reconsider the part of the compiler's transformation algorithm that determines whether argument extension shall take place or not. We would like to get rid of the heuristical approach by a formal model that estimates the runtime and synchronization costs as a more sound decision basis. It is also worth investigating whether and in which forms the task extension of function parameters (from inside the function body) can be propagated to an extension of the actual function arguments.

In spite of our critical comments on automatic parallelization, we also would like to investigate the automatic creation of para-functional annotations by the compiler assuming that the programmer has already made the key transformations. In this context, we would also like to introduce an extension of the annotation language where the programmer declares the runtime complexity of functions (as derived from the mathematical theory) to support the compiler in weight analysis and the decision where to create parallelism.

Bibliography

[ACM Lisp, 1992]
1992 ACM Conference on LISP and Functional Programming, San Francisco, California, June 22–24, 1992. ACM Press, New York.

[Appel and MacQueen, 1991]
Andrew Appel and David MacQueen.
Standard ML of New Jersey.
Release notes (version 0.75), AT&T Bell Laboratories, Murray Hill, New Jersey, November 1991.

[Arvind and Nikhil, 1987]
Arvind and R. S. Nikhil.
Executing a Program on the MIT Tagged-Token Dataflow Architecture.
In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE '87, Parallel Architectures and Languages Europe, Volume 2: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, June 15–19, 1987. Springer, Berlin.

[Arvind *et al.*, 1986]
Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali.
I-Structures: Data Structures for Parallel Computing.
In Joseph H. Fasel and Robert M. Keller, editors, *Graph Reduction, Proceedings of a Workshop*, volume 279 of *Lecture Notes in Computer Science*, pages 336–369, Santa Fe, New Mexico, September 29 – October 1, 1986. Springer, Berlin.

[Ashcroft and Wadge, 1977]
E. A. Ashcroft and W. W. Wadge.
Lucid, a Nonprocedural Language with Iteration.
Communications of the ACM, 20(7):519–526, July 1977.

[Banatre and Le Metayer, 1989]
Jean-Pierre Banatre and Daniel Le Metayer.
Chemical Reaction as a Computational Model.
In Kei Davis and John Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop*, Workshops in Computing, pages 103–117, Fraserburgh, Scotland, August 21–23, 1989. Springer, Berlin.

[Banatre and Le Metayer, 1990]
J.-P. Banatre and D. Le Metayer.
The GAMMA Model and its Discipline of Programming.
Science of Programming, 15(1):55–77, 1990.

[Barth *et al.*, 1991]
Paul S. Barth, Rishiyur S. Nikhil, and Arvind.

M-Structures: Extending a Parallel, Non-strict, Functional Language with State.
In John Hughes, editor, *Functional Programming Languages and Computer Architectures*, volume 523 of *Lecture Notes in Computer Science*, pages 538–568, Harvard, Massachusetts, 1991. Springer, Berlin.

[Bertoli *et al.*, 1994]
Pier Giorgio Bertoli, Hoon Hong, Andreas Neubacher, Wolfgang Schreiner, and Volker Stahl.
The Design of the STURM Distributed Multi-Processor Kernel.
Internal note, RISC-Linz, Johannes Kepler University, Linz, Austria, April 1994.

[Blelloch *et al.*, 1993]
Guy E. Blelloch, Siddharta Chatterjee, C. Hardwick Jonathan, Jay Sipelstein, and Marco Zagha.
Implementation of a Portable Nested Data-Parallel Language.
In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP*, pages 102–112, San Diego, California, May 19–22, 1993.

[Blelloch, 1993]
Guy E. Blelloch.
Nesl: A Nested Data-Parallel Language (Version 2.6).
Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[Bratvold, 1992]
Tore A. Bratvold.
Determining Useful Parallelism in Higher Order Functions.
In Herbert Kuchen and Rita Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, Aachen, Germany, September 28–30, 1992.

[Bratvold, 1993]
Tore A. Bratvold.
A Skeleton-Based Parallelising Compiler for ML.
In *5th International Workshop on the Implementation of Functional Languages*, pages 23–33, Numegen, The Netherlands, September, 1993.

[Bratvold, 1994]
Tore A. Bratvold.
Parallelising a Functional Program Using a List-Homomorphism Skeleton.
In Hoon Hong, editor, *PASCO 94 International Symposium on Parallel Symbolic Computation*, Hagenberg/Linz, Austria, September 26–28, 1994. World Scientific Publishing, Singapore.

[Buchberger and Hintenaus, 1985]
Bruno Buchberger and Peter Hintenaus.
The L-Language for the Parallel L-Machine — A Parallel Architecture for AI Applications.
In Trost and Retti, editors, *ÖGAI-Tagung 1985*, volume 106 of *Informatik Fachberichte*, pages 120–131, Vienna, Austria, September 24–27, 1985. Springer, Also: Technical Report 85-4, RISC-Linz, Johannes Kepler University, Linz, Austria, 1985.

[Buchberger *et al.*, 1982]
B. Buchberger, G. E. Collins, R. Loos, and R. Albrecht, editors.
Computer Algebra — Symbolic and Algebraic Computation.
Springer, Vienna, New York, 2nd edition, 1982.

[Buchberger *et al.*, 1992]
Bruno Buchberger, George Collins, Mark Encarnación, Hoon Hong, Jeremy Johnson, Werner Krandick, Rüdiger Loos, Ana Mandache, Andreas Neubacher, and Herbert Vielhaber.
A SACLIB Primer.
Technical Report 92-34, RISC-Linz, Johannes Kepler University, Linz, Austria, June 1992.

[Buchberger, 1978]
Bruno Buchberger.
Computer-Trees and their Programming.
In Dauchet, editor, *4th Colloquium "Trees in algebra and programming"*, pages 1–18, University of Lille, February 16–18, 1978. Also: Technical Report 78-11, RISC-Linz, Johannes Kepler University, Linz, Austria, 1978.

[Buchberger, 1985]
Bruno Buchberger.
The L-Machine: An Attempt at Parallel Hardware for Symbolic Computation.
In J. Calmet, editor, *3rd International Conference on Algebraic Algorithms and Error-Correcting Codes*, volume 229 of *Lecture Notes in Computer Science*, pages 333–347, Grenoble, France, July 15–19, 1985. Springer, Also: Technical Report 85-24, RISC-Linz, Johannes Kepler University, Linz, Austria, 1985.

[Buhr and Stroobosscher, 1990]
Peter A. Buhr and Richard A. Stroobosscher.
The μSystem: Providing Light-weight Concurrency on Shared-Memory Multiprocessor Computers Running UNIX.
Software — Practice and Experience, 20(9):929–964, September 1990.

[Bündgen *et al.*, 1994]
Reinhard Bündgen, Manfred Göbel, and Wolfgang Küchlin.
A Fine-Grained Parallel Completion Procedure.
In *ISSAC'94 — International Symposium on Symbolic and Algebraic Computation*, pages 269–277, Oxford, England, UK, July 20–22, 1994. ACM Press, New York.

[Burke-Perline, 1994]
Tracy Burke-Perline.
The Parallel Computation of $f(x)^{\frac{(p-1)}{2}} \bmod h(x)$ on a Network of Workstations using Sugarbush 1.1.
In Hoon Hong, editor, *PASCO 94 International Symposium on Parallel Symbolic Computation*, Hagenberg/Linz, Austria, September 26–28, 1994. World Scientific Publishing, Singapore.

[Burton, 1984]
F. Warren Burton.
Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs.
ACM Transactions on Programming Languages and Systems, 6(2):159–174, April 1984.

[Burton, 1987]
F. W. Burton.
Functional Programming for Concurrent and Distributed Computing.
The Computer Journal, 30(5):437–450, 1987.

[Burton, 1989]
F.Warren Burton.

Indeterminate Behavior with Determinate Semantics in Parallel Programs.
In *FPCA '89 Functional Programming Languages and Computer Architecture*, pages 340–346, London, UK, September 11–13, 1989. ACM Press, New York.

[Burton, 1991]
Warren F. Burton.
Encapsulating Non-Determinacy in an Abstract Data Type with Determinate Semantics.
Journal of Functional Programming, 1(1):3–20, January 1991.

[Busvine, 1993]
David John Busvine.
Detecting Parallel Structures in Functional Programs.
PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, April 1993.

[Cann, 1992]
D. C. Cann.
Retire Fortran? A Debate Rekindled.
Communications of the ACM, 35(8):81–89, August 1992.

[Carriero and Gelernter, 1990]
Nicholas Carriero and David Gelernter.
How to Write Parallel Programs.
MIT Press, 1990.

[Chandy and Misra, 1988]
Mani K. Chandy and Jayadev Misra.
Parallel Program Design — A Foundation.
Addison-Wesley, Reading, Massachusetts, 1988.

[Char *et al.*, 1994]
Bruce Char, Jeremy Johnson, David Saunders, and Andrew P. Wack.
Some Experiments with Parallel Bignum Arithmetic.
In Hoon Hong, editor, *PASCO 94 International Symposium on Parallel Symbolic Computation*, Hagenberg/Linz, Austria, September 26–28, 1994. World Scientific Publishing, Singapore.

[Char, 1990]
Bruce W. Char.
Progress Report on a System for General-Purpose Parallel Symbolic Algebraic Computation.
In *International Symposium on Symbolic and Algebraic Computation — ISSAC'90*, pages 96–103, Tokyo, Japan, August 20–24, 1990. ACM Press, New York.

[Chen *et al.*, 1991]
Marina Chen, Young-il Choo, and Jingke Li.
Crystal: Theory and Pragmatics of Generating Efficient Parallel Code.
In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 7, pages 255–308, ACM Press, New York, 1991.

[Chen, 1985]
Marina C. Chen.
Crystal: A Synthesis Approach to Programming Parallel Machines.
In Michael T. Health, editor, *First Conference on Hypercube Multiprocessors, Knoxville, Tennessee, August 26–27*, pages 87–107, 1985.

[Clarke, 1990]
Thomas Clarke.
The Semantics and Implementation of Aggregates or How to Express Concurrency without Destroying Determinism.
Technical report, Computer Laboratory, University of Cambridge, December 1990.

[Cole, 1988]
M. I. Cole.
A 'Skeletal' Approach to the Exploitation of Parallel Computation.
In *CONPAR 88*, volume C, pages 100–107, Manchester, UK, September 12–16, 1988. British Computer Society.

[Cole, 1989]
Murray I. Cole.
Algorithmic Skeletons: Structured Management of Parallel Computation.
Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.

[Collins and Loos, 19xx]
G. E. Collins and R. G. K. Loos.
SAC-2 System Documentation.
On-line documentation and program documentation, University of Tübingen, Germany and Ohio State University, Columbus, Ohio, 19xx.

[Crowl, 1994]
Lawrence A. Crowl.
How to Measure, Present, and Compare Parallel Performance.
IEEE Parallel and Distributed Technology, 2(1):9–25, Spring 1994.

[Danelutto *et al.*, 1994]
M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi.
Parallel Programming Models Based on the Restricted Computation Structure Approach.
In *ERCIM PPN Workshop on Parallel Processing Network*, pages 95–148, Heraklion, Crete, Greece, June 8–10, 1994. ICS-FORTH, Heraklion.

[Darlington *et al.*, 1991]
J. Darlington, A. J. Field, P. G. Harrison, D. Harper, G. K. Jouret, P. J . Kelly, K. M. Sephton, and D. W. Sharp.
Structured Parallel Functional Programming.
In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 31–52, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

[Darlington *et al.*, 1993]
J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu.
Parallel Programming Using Skeleton Functions.
In Arndt Bode, editor, *PARLE 93: Parallel Languages and Architectures Europe*, volume 694 of *Lecture Notes in Computer Science*, Munich, Germany, June 14–18, 1993. Springer, Berlin.

[Della Dora and Fitch, 1988]
J. Della Dora and J. Fitch, editors.
Computer Algebra and Parallelism, Workshop at the TIM3 Laboratory, University of Grenoble, France, June 1988. Academic Press, London.

[Feo and Oldehoeft, 1995]
John Feo and Livermore Oldehoeft, editors.

High Performance Functional Computing, Denver, Colorado, April, 9–11, 1995. To appear.

[Feo *et al.*, 1990]
J. T. Feo, D. C. Cann, and R. R. Oldehoeft.
A Report on the SISAL Language Project.
Journal of Parallel and Distributed Computing, 10(4):349–366, December 1990.

[Foster and Taylor, 1989]
Ian Foster and Stephen Taylor.
Strand — New Concepts in Parallel Programming.
Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[FPCA, 1993]
FPCA '93 Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, June 9–11, 1993. ACM Press, New York.

[Friedman and Wise, 1980]
Daniel Friedman and David Wise.
An Indeterminate Constructor for Applicative Programming.
In *7th Symposium on Principles of Programming Languages.* ACM Press, New York, 1980.

[Fuchssteiner *et al.*, 1993]
Benno Fuchssteiner, Waldemar Wiwianka, Klaus Gottheil, Andreas Kemper, Oliver Kluge, Karsten Morisse, Holger Naundorf, Gudrun Oevel, and Thorsten Schulze.
MuPAD — Multi Processing Algebra Data Tool (in German).
Birkhäuser, Basel, 1.1 edition, 1993.

[Fuchssteiner, 1992]
Benno Fuchssteiner.
MuPAD Reference Manual.
University of Paderborn, 1992, Version 1.0.

[Gautier and Roch, 1994]
Thierry Gautier and Jean-Louis Roch.
PAC++ System and Parallel Algebraic Numbers Computation.
In Hoon Hong, editor, *PASCO 94 International Symposium on Parallel Symbolic Computation*, Hagenberg/Linz, Austria, September 26–28, 1994. World Scientific Publishing, Singapore.

[Geist and Sunderam, 1992]
G. A. Geist and V. S. Sunderam.
Network-Based Concurrent Computing on the PVM System.
Concurrency, Practice and Experience, 4(4):293–311, June 1992.

[Gladitz and Kuchen, 1994]
Katia Gladitz and Herbert Kuchen.
Parallel Implementation of the Gamma-Operation on Bags.
In Hoon Hong, editor, *PASCO 94 International Symposium on Parallel Symbolic Computation*, Hagenberg/Linz, Austria, September 26–28, 1994. World Scientific Publishing, Singapore.

[Goldman and Gabriel, 1988]
Ron Goldman and Richard P. Gabriel.
Preliminary Results with the Initial Implementation of Qlisp.
In *ACM Conference on Lisp and Functional Programming*, pages 143–152, Snowbird, Utah, July 1988. ACM Press, New York.

[Goldman and Gabriel, 1989]
Ron Goldman and Richard P. Gabriel.
Qlisp: Parallel Processing in Lisp.
IEEE Software, pages 51–59, July 1989.

[Halstead, Jr., 1985]
Robert H. Halstead, Jr.
Multilisp: A Language for Concurrent Symbolic Computation.
ACM Transactions on Programming Languages and Systems, 7(4):501–538, October 1985.

[Halstead, Jr., 1986]
Robert H. Halstead, Jr.
Parallel Symbolic Computing.
IEEE Computer, 19(8):35–43, August 1986.

[Henderson, 1980]
Peter Henderson.
Functional Programming: Application and Implementation.
International Series in Computer Science, Prentice Hall, New Jersey, 1980.

[Henderson, 1982]
Peter Henderson.
Purely Functional Operating Systems.
In J. Darlington, Henderson P., and D. A. Turner, editors, *Functional Programming and its Applications*, pages 177–192. Cambridge University Press, 1982.

[Hill, 1993]
Jonathan M. D. Hill.
Data Parallel Haskell: Mixing Old and New Glue.
Technical report, Queen Mary and Westfield College, University of London, UK, 1993.

[Hintenaus, 1985]
Peter Hintenaus.
The L-Compiler and the L-Monitor. Two Tools for Programming the Parallel L-Machine.
Master's thesis, Johannes Kepler University, Linz, Austria, 1985, Also: Technical Report 86-2, RISC-Linz, Johannes Kepler University, Linz, Austria, 1986.

[Hoare, 1985]
C. A. R. Hoare.
Communicating Sequential Processes.
Series in Computer Science, Prentice Hall, Englewood Cliffs, NJ, 1985.

[Hong and Loidl, 1994]
Hoon Hong and Hans-Wolfgang Loidl.
Parallel Computation of Modular Multivariate Polynomial Resultants on Shared Memory Machine.
In Bruno Buchberger and Jens Volkert, editors, *Parallel Processing: CONPAR 94 — VAPP VI International Conference on Parallel and Vector Processing*, volume 854 of *Lecture Notes in Computer Science*, pages 325–336, Linz, Austria, September 6–8, 1994. Springer, Berlin, Also: Technical Report 94-19, RISC-Linz, Johannes Kepler University, Linz, Austria, March 1994.

[Hong and Schreiner, 1992]
Hoon Hong and Wolfgang Schreiner.

Programming in PACLIB.
SIGSAM Bulletin, 26(4):1–6, November 1992, Also: Technical Report 93-05, RISC-Linz, Johannes Kepler University, Linz, Austria, February 1993. Also: Technical Report ACPC/TR 93-13, Austrian Center for Parallel Computation, February 1993.

[Hong *et al.*, 1992]
Hoon Hong, Wolfgang Schreiner, Andreas Neubacher, Kurt Siegl, Hans-Wolfgang Loidl, Tudor Jebelean, and Peter Zettler.
PACLIB User Manual.
Technical Report 92-32, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1992, Also: Technical Report ACPC/TR 92-9, Austrian Center for Parallel Computation, July 1992.

[Hong *et al.*, 1993]
Hoon Hong, Andreas Neubacher, and Wolfgang Schreiner.
The Design of the SACLIB/PACLIB Kernels.
In Alfonso Miola, editor, *DISCO 93 — International Symposium on Design and Implementation of Symbolic Computation System*, volume 722 of *Lecture Notes in Computer Science*, pages 288–302, Gmunden, Austria, September 15–17, 1993. Springer, Berlin, Also: Technical Report 93-30, RISC-Linz, Johannes Kepler University, Linz, Austria, June 1993.

[Hong *et al.*, 1994a]
Hoon Hong, Andreas Neubacher, and Volker Stahl.
The STURM Library Manual — A C++ Library for Symbolic Computation.
Technical Report 94-30, RISC-Linz, Johannes Kepler University, Linz, Austria, April 1994.

[Hong *et al.*, 1994b]
Hoon Hong, Andreas Neubacher, Volker Stahl, and Wolfgang Schreiner.
The C++ Interface to the STURM Multi-Processor Kernel.
Technical Report 94-32, RISC-Linz, Johannes Kepler University, Linz, Austria, April 1994.

[Hong, 1991]
Hoon Hong.
Parallelization of Quantifier Elimination on Workstation Network.
Technical Report 91-55, RISC-Linz, Johannes Kepler University, Linz, Austria, September 1991.

[Hong, 1994]
Hoon Hong, editor.
PASCO 94 International Symposium on Parallel Symbolic Computation, Hagenberg/Linz, Austria, September 26–28, 1994. World Scientific Publishing, Singapore.

[Hudak and Anderson, 1987]
Paul Hudak and Steve Anderson.
Pomset Interpretations of Parallel Functional Programs.
In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 234–256, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

[Hudak and Smith, 1986]
Paul Hudak and Lauren Smith.
Para-Functional Programming: A Paradigm for Programming Multiprocessor Systems.

In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 243–254, St. Petersburg Beach, Florida, January 15–16, 1986. ACM Press, New York.

[Hudak *et al.*, 1992]
Paul [editor] Hudak, Simon [editor] Peyton Jones, Philip [editor] Wadler, Brian Boutel, Jon Fairbairn, Joeseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson.
Report on the Programming Language Haskell — A Non-strict, Purely Functional Language.
Research Report 1.2, Department of Computer Science and Department of Computing Science, Yale University, New Haven, Conneticut, U.S.A. and University of Glasgow, Glasgow, U.K., March 1992.

[Hudak, 1986a]
Paul Hudak.
Arrays, Non-Determinism, Side-Effects, and Parallelism: A Functional Perspective (Extended Abstract).
In Joseph H. Fasel and Robert M. Keller, editors, *Graph Reduction, Proceedings of a Workshop*, volume 279 of *Lecture Notes in Computer Science*, pages 312–327, Santa Fe, New Mexico, September 29 – October 1, 1986. Springer, Berlin.

[Hudak, 1986b]
Paul Hudak.
Para-Functional Programming.
IEEE Computer, 19(8):60–70, August 1986.

[Hudak, 1991]
Paul Hudak.
Para-Functional Programming in Haskell.
In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 5, pages 159–196, ACM Press, New York, 1991.

[Hughes and O'Donnell, 1989]
John Hughes and John O'Donnell.
Expressing and Reasoning about Non-deterministic Functional Programs.
In Kei Davis and John Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop*, Workshops in Computing, pages 287–295, Fraserburgh, Scotland, August 21–23, 1989. Springer, Berlin.

[ISSAC, 1994]
ISSAC'94 — International Symposium on Symbolic and Algebraic Computation, Oxford, England, UK, July 20–22, 1994. ACM Press, New York.

[Jayaraman and Plaisted, 1987]
Bharat Jayaraman and David A. Plaisted.
Functional Programming with Sets.
In Gilles Kahn, editor, *FPCA '87, Functional Programming Language and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 194–211, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

[Jouret, 1991]
Guido K. Jouret.
A Foundation for Declarative Data Parallelism.
In Hugh Glaser and Pieter Hartel, editors, *3rd International Workshop on the Parallel Implementation of Functional Languages*, pages 311–330, Southampton, UK, June 5–7, 1991. Technical Report CSTR 91-07, University of Southampton.

[Joy, 1990]
M. S. Joy.
Parallel Evaluation of Functional Programs — A Bibliography.
Functional Language Implementation Project Document 18, Department of Computer Science, University of Warwick, April 1990.

[Kass, 1986]
Peter Kass.
The L-Assembler — A Basic Software Tool for Programming the Parallel L-Machine.
Master's thesis, Johannes Kepler University, Linz, Austria, 1986, Also: Technical Report 86-4, RISC-Linz, Johannes Kepler University, Linz, Austria, 1986.

[Kelly, 1989]
Paul Kelly.
Functional Programming for Loosely-coupled Multiprocessors.
Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.

[Kemper *et al.*, 1991]
Andreas Kemper, Oliver Kluge, Karsten Morisse, Holger Naundorf, Waldemar Wianka, and Benno Fuchssteiner.
Computer Algebra on Multiprocessors.
Technical report, Department of Mathematics, University of Paderborn, March 1991.

[Kewley and Glynn, 1989]
John M. Kewley and Kevin Glynn.
Evaluation Annotations for Hope+.
In Kei Davis and John Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop*, Workshops in Computing, pages 329–337, Fraserburgh, Scotland, August 21–23, 1989. Springer, Berlin.

[Knuth, 1981]
Donald E. Knuth.
The Art of Computer Programming, Volume II: Seminumerical Algorithms.
Addison Wesley, Reading, 2nd edition, 1981.

[Kranz *et al.*, 1989]
David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr.
Mul-T: A High-Performance Parallel Lisp.
In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, number 24(7) in SIGPLAN Notices, pages 81–90, Portland, Oregon, June 21–23, July 1989. ACM Press, New York.

[Kredel, 1994]
Heinz Kredel.
Computeralgebra on a KSR1 Parallel Computer.
In *4th Supercomputing Day "Application of Supercomputers"*, Johannes Kepler University, Linz, February 3, 1994.

[Kröger, 1987]
Fred Kröger.
Temporal Logic of Programs, volume 8 of *EATCS Monographs on Theoretical Computer Science.*
Springer, Berlin, 1987.

[Kuchen and Gladitz, 1992]
Herbert Kuchen and Katia Gladitz.

Implementing Bags on a Shared Memory MIMD-Machine.
In Herbert Kuchen and Rita Loogen, editors, *4th International Workshop on the Parallel Implementation of Functional Languages*, Aachen, Germany, September 28–30, 1992.

[Küchlin and Nevin, 1991]
Wolfgang Küchlin and Nicholas Nevin.
On Multi-Threaded List-Processing and Garbage Collection.
In *3rd IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, December, 1991. IEEE Press.

[Küchlin and Ward, 1992]
Wolfgang Küchlin and Jeffrey Ward.
Experiments with Virtual C Threads.
In *4th IEEE Symposium on Parallel and Distributed Processing*, Arlington, Texas, December 1992. IEEE Press.

[Küchlin, 1990a]
Wolfgang Küchlin.
PARSAC-2: A Parallel SAC-2 Based on Threads.
In S. Sakata, editor, *Eighth International Symposium on Applied Algebra, Algebraic Algorithms, and Error Correcting Codes (AAECC-8)*, volume 508 of *Lecture Notes in Computer Science*, pages 206 –217, Tokyo, Japan, August 1990. Springer.

[Küchlin, 1990b]
Wolfgang Küchlin.
The S-Threads Environment for Parallel Symbolic Computation.
In R. E. Zippel, editor, *Computer Algebra and Parallelism: Second International Workshop on Parallel Algebraic Computation*, volume 584 of *Lecture Notes in Computer Science*, pages 1 – 18, Ithaca, USA, May 1990. Springer, Berlin.

[Limongelli and Loidl, 1993]
Carla Limongelli and Hans Wolfgang Loidl.
Rational Number Arithmetic by Parallel P-adic Algorithms.
In Jens Volkert, editor, *Parallel Computation — Second International ACPC Conference*, volume 734 of *Lecture Notes in Computer Science*, pages 72–86, Gmunden, Austria, October 4 – 6, 1993. Springer, Berlin, Also: Technical Report 93-41, RISC-Linz, Johannes Kepler University, Linz, Austria, July 1993.

[Limongelli and Pirastu, 1994]
Carla Limongelli and R. Pirastu.
Exact Solution of Linear Systems over Rational Numbers by Parallel p-adic Arithmetic.
In Bruno Buchberger and Jens Volkert, editors, *Parallel Processing: CONPAR 94 — VAPP VI International Conference on Parallel and Vector Processing*, volume 854 of *Lecture Notes in Computer Science*, pages 313–324, Linz, Austria, September 6–8, 1994. Springer, Berlin.

[Limongelli, 1993]
Carla Limongelli.
On an Efficient Algorithm for Big Rational Number Computations by Parallel p-adic.
Journal of Symbolic Computation, 15(2):181–197, February 1993.

[Loidl, 1993]
Hans Wolfgang Loidl.
Solving a System of Linear Equations by Using a Modular Method.
Technical Report 93-69, RISC-Linz, Johannes Kepler University, Linz, Austria, December 1993.

[Manna and Pnueli, 1981a]
Zohar Manna and Amir Pnueli.
Verification of Concurrent Programs, Part I: The Temporal Framework.
Technical Report STAN–CS–81–836, Department of Computer Science, Stanford University, Stanford, CA, June 1981.

[Manna and Pnueli, 1981b]
Zohar Manna and Amir Pnueli.
Verification of Concurrent Programs, Part II: Temporal Proof Principles.
Technical Report STAN–CS–81–843, Department of Computer Science, Stanford University, Stanford, CA, September 1981.

[Maranget, 1991]
Luc Maranget.
GAML: a Parallel Implementation of Lazy ML.
In John Hughes, editor, *FPCA '91, Functional Programming Languages and Computer Architectures*, volume 523 of *Lecture Notes in Computer Science*, pages 102–123, Harvard, Massachusetts, USA, 1991. Springer, Berlin.

[Marino and Succi, 1989]
Giuseppe Marino and Giancarlo Succi.
Data Structures for Parallel Execution of Functional Languages.
In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE '89, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures*, volume 365 of *Lecture Notes in Computer Science*, pages 346–356, Eindhoven, The Netherlands, June 12–16, 1989. Springer, Berlin.

[McCarthy, 1963]
John McCarthy.
A Basis for a Mathematical Theory of Computation.
In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[Merrall, 1991a]
Simon Merrall.
Plurals — A SIMD extension to Eulisp.
Technical report, School of Mathematical Sciences, University of Bath, 1991.

[Merrall, 1991b]
Simon Merrall.
TPL — An Implementation of Paralation Lisp in EuLisp.
Technical report, School of Mathematical Sciences, University of Bath, 1991.

[Michaelson *et al.*, 1993]
Greg Michaelson, Tore Bratvold, David Busvine, and Kevin Waugh.
Parallel Implementations from Functional Prototype Instrumentation.
In *PARCO' 93*, Grenoble, France, September, 1993. Elsevier North-Holland.

[Miller, 1987]
J. S. Miller.
MultiScheme, A Parallel Processing System Based on MIT Scheme.
PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1987, Also: Technical Report MIT LCS/TR/402.

[Milner, 1980]
Robin Milner.

A Calculus of Communicating Systems, volume 92 of *Lecture Notes in Computer Science.*
Springer, 1980.

[Milner, 1993]
Robert Milner.
Elements of Interaction.
Communications of the ACM, 36(1):78–89, January 1993.

[Minimair, 1992]
Manfred Minimair.
Computation of the Q-Matrix.
Internal report, RISC-Linz, Johannes Kepler University, Linz, Austria, June 1992.

[Miola, 1993]
Alfonso Miola, editor.
DISCO'93 — Design and Implementation of Symbolic Computation Systems, volume 722 of *Lecture Notes in Computer Science*, Gmunden, Austria, September 15–17, 1993.
Springer, Berlin.

[Mohr *et al.*, 1990]
Eric Mohr, David A. Kranz, and Robert H. Halstead Jr.
Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs.
In *1990 ACM Symposium on Lisp and Functional Programming*, pages 185–197, Nice, France, June 27–29, 1990. ACM Press.

[Mohr *et al.*, 1991]
Eric Mohr, David A. Kranz, and Robert H. Halstead Jr.
Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs.
IEEE Transactions on Parallel and Distributed Systems, 2(3):264–280, July 1991.

[Nikhil *et al.*, 1993]
Rishiyur S. Nikhil, Arvind, and James Hicks.
pH Language Proposal.
Preliminary language reference manual, DEC Cambridge Research Lab, MIT Lab for Computer Science, Motorola Cambridge Research Lab, September 1993.

[Nikhil, 1991]
R. S. Nikhil.
Id (Version 90.1) Reference Manual.
CSG Memo 284-2, MIT Laboratory for Computer Science, Cambridge, MA, July 1991.

[Nöcker *et al.*, 1991]
E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. Eekelen, and M. J. Plasmeijer.
Concurrent Clean.
In E. H. L. Aarts, J. van Leeuwen, and M. Rem, editors, *PARLE '91, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures and Algorithms*, volume 505 of *Lecture Notes in Computer Science*, pages 202–219, Eindhoven, The Netherlands, June 10–13, 1991. Springer, Berlin.

[Oldehoeft and Cann, 1988]
R. R. Oldehoeft and D. C. Cann.
Applicative Parallelism on a Shared-Memory Multiprocessor.
IEEE Software, 5(1):62–70, January 1988.

[Ossefort, 1983]
Marty Ossefort.

Correctness Proofs of Communicating Processes: Three Illustrative Examples from the Literature.
ACM Transactions on Programming Languages and Systems, 5(4):620–640, October 1983.

[Owicki and Gries, 1976]
S. Owicki and D. Gries.
Verifying Properties of Parallel Programs: An Axiomatic Approach.
Communications of the ACM, 19(5):279–285, May 1976.

[Owicki and Lamport, 1982]
S. Owicki and L. Lamport.
Proving Liveness Properties of Parallel Programs: An Axiomatic Approach.
ACM Transactions on Programming Languages and Systems, 4(3):455–495, July 1982.

[Pehoushek and Weening, 1988]
Joseph D. Pehoushek and Joseph S. Weening.
Low-Cost Process Creation and Dynamic Partitioning in Qlisp.
In T. Ito and R. H. Halstead, editors, *Parallel Lisp: Languages and Systems*, volume 441 of *Lecture Notes in Computer Science*, pages 182–199, Sendai, Japan, June 5–8, 1988. Springer, Berlin.

[Ponder, 1988a]
C. G. Ponder.
Evaluation of "Performance Enhancements" in Algebraic Manipulation Systems.
Technical Report UCB/CSD 88/438, Computer Science Division (EECS), University of California, August 1988.

[Ponder, 1988b]
C. G. Ponder.
Parallel Processors and Systems for Algebraic Manipulation: Current Work.
ACM SIGSAM Bulletin, 22(3):15–21, July 1988.

[Ponder, 1988c]
C. G. Ponder.
Parallelism and Algorithms for Algebraic Manipulation: Current Work.
ACM SIGSAM Bulletin, 22(3):7–14, July 1988.

[POPL, 1994]
POPL '94: 21st Annual Symposium on Principles of Programming Languages, Portland, Oregon, January 16–19, 1994. ACM Press, New York.

[PPoPP, 1993]
Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, May 19-22, 1993. ACM Press, New York.

[Rabhi, 1993a]
F. A. Rabhi.
Exploiting Parallelism in Functional Languages: A "Paradigm-Oriented" Approach.
In T. Lake and P. Dew, editors, *Abstract Machine Models for Highly Parallel Computers*, Oxford University Press, 1993.

[Rabhi, 1993b]
Fethi A. Rabhi.
Functional Languages and Parallelism.
In *3rd Workshop on Parallel Computing*, Polytechnique de Mons, Belgium, September 23–24, 1993.

[Richards Jr., 1984]
Hamilton Richards Jr.
An Applicative Programming Languages Bibliography.
Technical report, Burroughs Corporation, Austin Research Center, Austin, Texas, October 1984.

[Roch, 1988]
J. L. Roch.
PAC: Towards a Parallel Computer Algebra Co-Processor.
In J. Della Dora and J. Fitch, editors, *Computer Algebra and Parallelism*, pages 33–50, Workshop at the TIM3 Laboratory, University of Grenoble, France, June 1988. Academic Press, London.

[Roch, 1990]
J. L. Roch.
An Environment for Parallel Algebraic Computation.
In R. E. Zippel, editor, *Computer Algebra and Parallelism, Second International Workshop on Parallel Algebraic Computation*, volume 584 of *Lecture Notes in Computer Science*, pages 33–50, Ithaca, USA, May 1990. Springer, Berlin.

[Sabot, 1988]
Gary Sabot.
The Paralation Model: Architecture Independent SIMD Programming.
MIT Press, Cambridge, MA, 1988.

[Sannella, 1994]
Donald Sannella, editor.
ESOP 94, 5th European Symposium on Programming, volume 788 of *Lecture Notes in Computer Science*, Edinburgh, UK, April 11–13, 1994. Springer, Berlin.

[Schmidt, 1986]
David A. Schmidt.
Denotational Semantics — A Methodology for Language Development.
Allyn and Bacon, Boston, 1986.

[Schreiner and Hong, 1993a]
Wolfgang Schreiner and Hoon Hong.
A New Library for Parallel Algebraic Computation.
In Richard F. Sincovec et al., editors, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, volume II, pages 776–783, Norfolk, Virginia, March 22–24, 1993. SIAM, Also: Technical Report 92-73, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992. Also: Technical Report ACPC/TR 93-9, Austrian Center for Parallel Computation, February 1993.

[Schreiner and Hong, 1993b]
Wolfgang Schreiner and Hoon Hong.
PACLIB — A System for Parallel Algebraic Computation on Shared Memory Computers.
In Hussein M. Alnuweiri, editor, *Parallel Systems Fair at the Seventh International Parallel Processing Symposium*, pages 56–61, Newport Beach, CA, April 14, 1993. IPPS '93, Also: Technical Report 93-03, RISC-Linz, Johannes Kepler University, Linz, Austria, February 1993. Also: Technical Report ACPC/TR 93-12, Austrian Center for Parallel Computation, February 1993.

[Schreiner and Hong, 1993c]
Wolfgang Schreiner and Hoon Hong.

The Design of the PACLIB Kernel for Parallel Algebraic Computation.
In Jens Volkert, editor, *Parallel Computation — Second International Conference of the Austrian Center for Parallel Computation*, volume 734 of *Lecture Notes in Computer Science*, pages 204–218, Gmunden, Austria, October 4–6, 1993. Springer, Berlin.

[Schreiner, 1992]
Wolfgang Schreiner.
The Design of the PACLIB Kernel.
Technical Report 92-33, RISC-Linz, Johannes Kepler University, Linz, Austria, July 1992, Also: Technical Report ACPC/TR 93-4, ACPC Technical Report Series, Austrian Center for Parallel Computation, February 1993.

[Schreiner, 1993a]
Wolfgang Schreiner.
Compiling a Functional Language to Efficient SACLIB C.
Technical Report 93-49, RISC-Linz, Johannes Kepler University, Linz, Austria, September 1993.

[Schreiner, 1993b]
Wolfgang Schreiner.
Parallel Functional Programming — An Annotated Bibliography (2nd Edition).
Technical Report 93-24, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1993.

[Schreiner, 1993c]
Wolfgang Schreiner.
The Correctness of the PACLIB Kernel — A Case Study in Parallel Program Verification by Temporal Logic.
Technical Report 93-13, RISC-Linz, Johannes Kepler University, Linz, Austria, March 1993, Also: Technical Report ACPC/TR 93-15, Austrian Center for Parallel Computation, June 1993.

[Schreiner, 1994a]
Wolfgang Schreiner.
A Para-Functional Programming Interface for a Parallel Computer Algebra Package.
In Hoon Hong, editor, *PASCO 94 International Symposium on Parallel Symbolic Computation*, Hagenberg/Linz, Austria, September 26–28, 1994. World Scientific Publishing, Singapore, Also: Technical Report 94-46, RISC-Linz, Johannes Kepler University, Linz, Austria, August 1994.

[Schreiner, 1994b]
Wolfgang Schreiner.
Compiling a Para-Functional Language to Parallel PACLIB C.
Technical Report 94-44, RISC-Linz, Johannes Kepler University, Linz, Austria, July 1994.

[Schreiner, 1994c]
Wolfgang Schreiner.
Virtual Tasks for the PACLIB Kernel.
In Bruno Buchberger and Jens Volkert, editors, *Parallel Processing: CONPAR 94 — VAPP VI International Conference on Parallel and Vector Processing*, volume 854 of *Lecture Notes in Computer Science*, pages 533–544, Linz, Austria, September 6–8, 1994. Springer, Berlin.

[Seitz, 1988]
Steffen Seitz.

Parallel Algorithm Development.
In J. Della Dora and J. Fitch, editors, *Computer Algebra and Parallelism*, pages 223 –232, Workshop at the TIM3 Laboratory, University of Grenoble, France, June 1988. Academic Press, London.

[Seitz, 1990]
Steffen Seitz.
Algebraic Computation on a Local Net.
In R. E. Zippel, editor, *Computer Algebra and Parallelism, Second International Workshop on Parallel Algebraic Computation*, volume 584 of *Lecture Notes in Computer Science*, pages 19–31, Ithaca, USA, May 1990. Springer, Berlin.

[Siegl, 1992]
Kurt Siegl.
||MAPLE||: A System for Parallel Computer Algebra for Transputers.
In Szabolcs Ferenczi, editor, *1st Austrian-Hungarian Workshop on Transputer Applications*, number KFKI-1992-34/M,N in Technical Report Series of the Hungarian Academy of Sciences Central Research Institute for Physics, pages 21–30, Sopron, Hungary, October 8–10, 1992. Hungarian Transputers User Group and Austrian Center for Parallel Computation, Also: Technical Report 93-06, RISC-Linz, Johannes Kepler University, Linz, Austria, February 1993.

[Siegl, 1993a]
Kurt Siegl.
||MAPLE|| — A System for Parallel Symbolic Computation.
In Hussein M. Alnuweiri, editor, *Parallel Systems Fair at the Seventh International Parallel Processing Symposium*, pages 62–67, Newport Beach, California, April 14, 1993. IPPS '93, Also: Technical Report 93-07, RISC-Linz, Johannes Kepler University, Linz, Austria, February 1993.

[Siegl, 1993b]
Kurt Siegl.
Parallelizing Algorithms for Symbolic Computation Using ||MAPLE||.
In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–186, San Diego, California, May 19-22, 1993. ACM Press, Also: Technical Report 93-08, RISC-Linz, Johannes Kepler University, Linz, Austria, February 1993.

[SIGPLAN, 1994]
SIGPLAN '94: 7th Conference on Programming Language, Design and Implementation, volume 29(6) of *SIGPLAN Notices*, Orlando, Florida, June 20–24, 1994. ACM Press, New York.

[Søndergaard and Sestoft, 1988]
H. Søndergaard and P. Sestoft.
Non-Determinism in Functional Languages.
Technical Report 88-18, Department of Computer Science, University of Melbourne, Australia, 1988.

[Stahl, 1992]
Volker Stahl.
Solving Systems of Linear Equations with Modular Arithmetic on a MIMD Computer.
Technical Report 92-62, RISC-Linz, Johannes Kepler University, Linz, Austria, October 1992, Also: Technical Report ACPC/TR 93-5, Austrian Center for Parallel Computation, February 1993.

[Steele Jr. and Hillis, 1986]
Guy L. Steele Jr. and W. Daniel Hillis.
Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing.
In *1986 ACM Conference on Lisp and Functional Programming*, pages 279–297, Cambridge, Massachusetts, August 4–6, 1986. ACM Press, New York.

[Szymanski, 1987]
B. Szymanski.
Parallel Programming with Recurrent Equations.
International Journal of Supercomputing Applications, 1(2):74–74, 1987.

[Szymanski, 1991]
Boleslaw K. Szymanski.
EPL — Parallel Programming with Recurrent Equations.
In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 3, pages 51–104, ACM Press, New York, 1991.

[Tan and Wang, 1988]
Trevor B. Tan and Paul S. Wang.
Automatic Generation of Parallel Code for the Warp Computer.
In *Computer Algebra and Parallelism*, pages 91–117, Workshop at the TIM3 Laboratory, University of Grenoble, France, June 1988. Academic Press, London.

[Turner, 1987]
David Turner.
Functional Programming and Communicating Processes.
In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE '87 Parallel Architectures and Languages Europe, Volume 2: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, pages 54–74, Eindhoven, The Netherlands, June 15–19, 1987. Springer, Berlin.

[Wadge and Ashcroft, 1985]
William W. Wadge and Edward A. Ashcroft.
Lucid, the Dataflow Programming Language.
Number 22 in A.P.I.C. Studies in Data Processing, Academic Press, London, 1985.

[Wadler, 1991]
Philip Wadler.
Is There a Use for Linear Logic?
In *ACM/IFIP Symposium on Partial Evaluation and Semantics Based Program Manipulation (PEPM)*, Yale University, June 1991.

[Wang, 1991]
Paul S. Wang.
Symbolic Computation and Parallel Software.
In H. P. Zima, editor, *Parallel Computation — First International ACPC Conference*, volume 591 of *Lecture Notes in Computer Science*, pages 316–337. Springer, Berlin, 1991.

[Wang, 1992]
Paul S. Wang.
Parallel Univariate p-adic Lifting on Shared-Memory Multiprocessors.
In Paul S. Wang, editor, *ISSAC'92 — International Symposium on Symbolic and Algebraic Computation*, pages 168 – 176, Berkeley, California, July 27 – 29, 1992. ACM Press.

[Watt, 1986]
Stephen Michael Watt.
Bounded Parallelism in Computer Algebra.
Research Report CS-86-12, Faculty of Mathematics, University of Waterloo, Ontario, Canada, 1986.

[Willis and Bogucz, 1993]
T. J. Willis and E. A. Bogucz.
Coupling Symbolic Processing with Parallel Numeric Computation.
In Richard F. Sincovec, David E. Keyes, R. Leuze, Michael, Linda R. Petzold, and Daniel A. Reed, editors, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 788–792, Norfolk, Virginia, March 22–24, 1993. SIAM, Philadelphia.

[Workshop, 1993]
5th International Workshop on the Parallel Implementation of Functional Languages, Numegen, The Netherlands, September, 1993.

[Zippel, 1990]
R. E. Zippel, editor.
Computer Algebra and Parallelism, Proceedings of the Second International Workshop on Parallel Algebraic Computation, volume 584 of *Lecture Notes in Computer Science*, Ithaca, USA, May 1990. Springer, Berlin.

Appendix A

The Correctness Lemmas

We now list all auxiliary lemmas classified into the following categories:

1. **Mutual Exclusion Lemmas** restricting the access to program regions and data structures.
2. **Operation Lemmas** describing the validity and the effects of several program operations.
3. **Structure Lemmas** describing properties that can be derived from the static structure of the program.
4. **Main Lemmas** that essentially capture the dynamic behavior of the program.
5. **Continuation Lemmas** that describe eventuality properties of the program.

The lemmas are listed in a bottom-up order i.e. the proof of a lemma is based only on previously listed lemmas (with some explicitly mentioned exceptions).

A.1 Mutual Exclusion Lemmas

We define the predicate $\texttt{invar}(a)$ by

$$\texttt{invar}(a) \equiv \forall \overline{a}(a = \overline{a} \rightarrow \bigcirc a = \overline{a})$$

saying "a is not changed in the current time moment" and the propositional operator $\texttt{exor}(\{A_1, \ldots, A_n\})$

$$\begin{array}{rl} \texttt{exor}(\{A_1, \ldots, A_n\}) \equiv & (A_1 \wedge \neg A_2 \wedge \ldots \wedge \neg A_n) \vee \\ & (\neg A_1 \wedge A_2 \wedge \ldots \wedge \neg A_n) \vee \\ & \ldots \\ & (\neg A_1 \wedge \neg A_2 \wedge \ldots \wedge A_n) \end{array}$$

saying "exactly one of $A_1, \ldots, A_n$ is true".

The following lemma declares a set of program variables to be "local".

Lemma 17 (Locality) *The variables w_i, S_i, s_i, $value_i$, $avail_i$, $deliver_i$, P_j, p_j, Q_j, q_j and $found_j$ are only modified by the processes i and j, respectively:*

$$\tau_\Pi \to \Box\forall i \forall i' \forall l((\lambda_l^{i'} \land i' \neq i) \to (\mathtt{invar}(w_i) \land \mathtt{invar}(S_i) \land \mathtt{invar}(s_i) \land \mathtt{invar}(value_i) \land \mathtt{invar}(avail_i) \land \mathtt{invar}(deliver_i)))$$

$$\tau_\Pi \to \Box\forall j \forall j' \forall l((\lambda_l^{j'} \land j' \neq j) \to (\mathtt{invar}(P_j) \land \mathtt{invar}(p_j) \land \mathtt{invar}(Q_j) \land \mathtt{invar}(q_j) \land \mathtt{invar}(found_j)))$$

Proof: These variables are only modified in $result_i$ and $wait_j$, respectively. □

The following lemma declares the mutual exclusion of several critical regions protected by the semaphore $wsem_i$. If $wsem_i$ is true, no process is within the denoted regions; if $wsem_i$ is false, exactly one process is in one of these regions. Please not that only one critical regions is "statically defined" by the structure of the program (i.e. by the value of the location counter of a process), while the others depend on the contents of the program variables $s_{i'}$ and p_j, respectively.

Lemma 18 (Wait Semaphore) *The family of semaphores $wsem_i$ establishes the mutual exclusion of several critical regions:*

$$\begin{aligned}\tau_\Pi \to \Box\forall i(\mathtt{exor}\{ & \\ & wsem_i, \\ & \mathtt{at}\{\alpha_4^i \ldots \alpha_6^i, \alpha_9^i, \alpha_{10}^i, \alpha_{24}^i\}, \\ & s_1 = i \land \mathtt{at}\{\alpha_{16}^1, \alpha_{17}^1\}, \\ & \ldots \\ & s_n = i \land \mathtt{at}\{\alpha_{16}^n, \alpha_{17}^n\}, \\ & p_1 = i \land \mathtt{at}\{\beta_6^1 \ldots \beta_8^i, \beta_{10}^i\}, \\ & \ldots \\ & p_n = i \land \mathtt{at}\{\beta_6^n \ldots \beta_8^n, \beta_{10}^n\}, \\ & q_1 = i \land \mathtt{at}\{\beta_{18}^1 \ldots \beta_{19}^1\}, \\ & \ldots \\ & q_n = i \land \mathtt{at}\{\beta_{18}^n \ldots \beta_{19}^n\}\}).\end{aligned}$$

Proof: Proving this "obvious" program property is a basically simple but practically tedious task. By Lemma 2 (Invariant Rule), it suffices to show that mutual exclusion holds initially (which is a consequence of start_Π) and that it is invariant under each action λ.

Since s_i, p_j and q_j are local to process i and j respectively and s_i, p_j and q_j are not changed within the critical regions, it suffices to check the invariance of the property at the entries and the exits of the critical regions:

1. $\lambda \in \{\alpha_3^i, \alpha_6^i, \alpha_8^i, \alpha_{10}^i, \alpha_{23}^i, \alpha_{24}^i\}$,
2. $\lambda \in \{\alpha_{15}^{i'}, \alpha_{17}^{i'}\}$ assuming $s_{i'} = i$,
3. $\lambda \in \{\beta_5^j, \beta_8^j, \beta_{10}^j\}$ assuming $p_j = i$ and
4. $\lambda \in \{\beta_{17}^j, \beta_{19}^j\}$ assuming $p_j = i$.

The details are left to the reader. □

The previous lemma only describes the low-level effect of the semaphore $wsem_i$; its high-level purpose (the protection of the set W_i) is actually expressed by the following lemma.

Lemma 19 (Wait Exclusion) *Each wait set W_i is only modified in a critical region protected by $wsem_i$:*

$$\tau_\Pi \to \Box\forall i(wsem_i \to \mathtt{invar}(W_i)).$$

Proof: This is a corollary of Lemma 18 (Wait Semaphore) (check!). □

Likewise, we define the effect and the purpose of the semaphore $rsem_j$.

Lemma 20 (Result Semaphore) *The family of semaphores $rsem_j$ establishes the mutual exclusion of several critical regions:*

$$\begin{aligned}
&\tau_\Pi \rightarrow \Box\forall j(\mathtt{exor}\{\\
&\quad rsem_j,\\
&\quad \mathtt{at}\{\beta_3^j \ldots \beta_{21}^j\},\\
&\quad w_1 = j \wedge \mathtt{at}\{\alpha_8^1 \ldots \alpha_{22}^1\},\\
&\quad \ldots\\
&\quad w_n = j \wedge \mathtt{at}\{\alpha_8^n \ldots \alpha_{22}^n\}\}).
\end{aligned}$$

Proof: The proof proceeds analogously to the proof of Lemma 18 (Wait Semaphore) by checking that mutual exclusion holds initially (which is a direct consequence of A) and that this property is invariant under the following actions λ:

1. $\lambda \in \{\beta_2^j, \beta_{21}^j\}$ and
2. $\lambda \in \{\alpha_7^i, \alpha_{22}^i\}$ (assuming $w_i = j$).

The details are left to the reader. □

Lemma 21 (Result Exclusion) *The following actions are only performed within a critical region protected by $rsem_j$:*

1. *y_j is modified.*
2. *r_j is modified.*
3. *run_j is set to true.*
4. *j is added to or removed from some W_i.*

Formally,

$$\begin{aligned}
&\tau_\Pi \rightarrow \Box(rsem_j \rightarrow\\
&\quad \mathtt{invar}(y_j) \ \wedge\ \mathtt{invar}(r_j) \ \wedge \mathtt{invar}(run_j) \ \wedge\\
&\quad \forall i(\mathtt{elem}_i(j, W_i) \leftrightarrow \bigcirc\mathtt{elem}_i(j, W_i))).
\end{aligned}$$

Proof: This is a corollary of Lemma 20 (Result Semaphore), Lemma 18 (Wait Semaphore) and the program structure. □

A.2 Operation Lemmas

The effect of the operations **add** and **rem** on W_i is described by the following lemma.

Lemma 22 (Set Update) *If W_i is updated by adding or removing some j, then the update has achieved a consistent state at the end of the corresponding critical region:*

$$\begin{aligned}
&\Box\forall i\forall j\forall k\forall\overline{W}(\\
&\quad ((\alpha_{15}^i \wedge W_i = \overline{W} \wedge w_i = j) \rightarrow (W_i = \overline{W}\backslash\{j\} \ \mathtt{atnext}\ \alpha_{17}^i)) \wedge\\
&\quad ((\beta_5^j \wedge p_j = k \wedge W_k = \overline{W} \wedge \neg avail_k) \rightarrow (W_k = \overline{W} \cup \{j\} \ \mathtt{atnext}\ \beta_8^j)) \wedge\\
&\quad ((\beta_{17}^j \wedge q_j = k \wedge W_k = \overline{W}) \rightarrow (W_k = \overline{W}\backslash\{j\} \ \mathtt{atnext}\ \beta_{19}^j))).
\end{aligned}$$

Proof: By Lemma 18 (Wait Semaphore). □

The following lemma guarantees that we are allowed to use the $\in$ assignment in the program model.

Lemma 23 (Element) *The $\in$ assignment is well-defined:*

$$\tau_\Pi \rightarrow \Box\forall i((\alpha_5^i \rightarrow W_i \neq \emptyset) \wedge (\alpha_{14}^i \rightarrow S_i \neq \emptyset)),$$
$$\tau_\Pi \rightarrow \Box((\beta_4^j \rightarrow P_j \neq \emptyset) \wedge (\beta_{16}^j \rightarrow Q_j \neq \emptyset)).$$

Proof: Lemma 18 (Wait Semaphore) and Axiom 14 (While Loop) imply

$$\tau_\Pi \rightarrow \Box\forall i(\alpha_5^i \rightarrow W_i \neq \emptyset).$$

The rest is a consequence of Lemma 17 (Locality) and Axiom 14 (While Loop). □

A.3 Structure Lemmas

The following lemma tells us that in certain contexts we need not distinguish between x_i and $\bar{x}_i$ and between R_j and $\bar{R}_j$:

Lemma 24 (Constancy) *The parameters x_i and R_j are not changed in $result_i$ and $wait_j$, respectively:*

$$\tau_\Pi \rightarrow \Box\forall i(\alpha_1^i \rightarrow \Box(x_i = \overline{x}_i)) \wedge$$
$$\tau_w \rightarrow ((R_j = \overline{R}_j \ \texttt{until}\ \tau_r) \wedge (R_j = \overline{R}_j \ \texttt{atnext}\ \tau_r)).$$

Proof: By the structure of the program. □

The following lemmas describe the state of several program variables at certain program locations.

Lemma 25 (Available) *After the available flag is set, it remains true in $result_i$:*

$$\tau_\Pi \rightarrow \Box\forall i(\ \texttt{at}\{\alpha_3^i \ldots \alpha_e^i\} \rightarrow avail_i).$$

Proof: By Lemma 17 (Locality). □

Lemma 26 (If Available) *In the else branch of $wait_j$, $avail_{p_j}$ always holds:*

$$\tau_\Pi \rightarrow \Box\forall j(\ \texttt{at}\{\beta_{10}^j \ldots \beta_{20}^j\} \rightarrow avail_{p_j}).$$

Proof: By Axiom 12 (Two Branch Conditional) and Lemma 17 (Locality). □

Lemma 27 (Not Available) *If process i does not deliver a result, $avail_i$ will never hold:*

$$\tau_\Pi \rightarrow \forall i(\Box\neg\ \texttt{at}\{\alpha_1^i\} \rightarrow \Box\neg avail_i).$$

Proof: By Rule 4 (Induction). □

Lemma 28 (Process Set) *Within $wait_j$, the variables p_j and r_j are always members of the process set $\overline{R}_j$:*

$$\tau_\Pi \rightarrow \Box((\ \texttt{at}\{\beta_5^j \ldots \beta_{20}^j\} \rightarrow p_j \in \overline{R}_j) \wedge (\ \texttt{at}\{\beta_{17}^j \ldots \beta_{20}^j\} \rightarrow q_j \in \overline{R}_j).$$

Proof: By Rule 4 (Induction). □

Lemma 29 (Result) *If the available flag is set, then the result is indeed available:*

$$\tau_\Pi \rightarrow \Box\forall i(avail_i \rightarrow value_i = \overline{x}_i).$$

Proof: By Rule 24 (Constancy) using that only α_2^i sets $avail_i$ to true. □

Lemma 30 (Found) *After $found_j$ is set to true, it remains true:*

$$\tau_\Pi \rightarrow \Box(\ \texttt{at}\{\beta_{14}^j \ldots \beta_{20}^j\} \rightarrow found_j).$$

Proof: By Lemma 17 (Locality) and the structure of t eh program. □

Lemma 31 (Not Found) *Before $found_j$ is set to true, $found_j$ is false:*

$$\tau_\Pi \rightarrow \Box(\texttt{at}\{\beta^j_4 \ldots \beta^j_{13}\} \rightarrow \neg found_j).$$

Proof: By Axiom 14 (While Loop) and Lemma 17 (Locality). □

Lemma 32 (Reset) *If $found_j$ is set to false, it was true before:*

$$\tau_\Pi \rightarrow \Box(\texttt{at}\{\beta^j_{23}\} \rightarrow found_j).$$

Proof: By Axiom 12 (Two Branch Conditional) and Lemma 17 (Locality). □

Lemma 33 (Sleeping) *If j is going to sleep, $found_j$ is false.*

$$\tau_\Pi \rightarrow \Box(\texttt{at}\{\beta^j_{24}\} \rightarrow \neg found_j).$$

Proof: By Axiom 12 (Two Branch Conditional) and Lemma 17 (Locality). □

Lemma 34 (Deliver) *If process i is in the "then" branch of $result_i$, then $deliver_i$ was set at the previous α^i_9:*

$$\tau_\Pi \rightarrow \Box\forall i(\texttt{at}\{\alpha^i_{12} \ldots \alpha^i_{21}\} \rightarrow (deliver_i \wedge w_i \in W_i \ \texttt{atprev}\ \alpha^i_9)).$$

Proof: This is a consequence of Axiom 13 (One Branch Conditional) and Lemma 17 (Locality). □

A.4 Main Lemmas

In this section, we give the key lemmas that capture the dynamic behavior of the program. The organization of these lemmas is as follows:

1. **Entry:** Lemma 35 describes the state of Π when $wait_j$ is entered (i.e. it is of the form $\tau_w \rightarrow A$).
2. **Others:** Most lemmas capture the behavior of Π during the execution of $wait_j$ (i.e. they are of the form $\tau_w \rightarrow (A\ \texttt{until}\ \tau_r)$).
3. **Exit:** Lemma 46 describes the state of Π immediately before $wait_j$ is left (i.e. it is of the form $\tau_f \rightarrow (A\ \texttt{until}\ \tau_r)$.

For proving these lemmas, we perform an induction over the number of times that $wait_j$ has been executed before:

1. **Induction Base:** We prove that Lemma 35 holds when $wait_j$ is entered for the first time i.e. has not been executed before.
2. **Others:** Based on the induction hypothesis Lemma 35, we prove that the other lemmas hold.
3. **Exit:** Based on the other lemmas, we prove that Lemma 46 holds.
4. **Induction Step:** based on Lemma 46, we prove that Lemma 35 holds when $wait_j$ is entered for the next time.

Both the induction base and the induction step are given in the proof of Lemma 35 (Entry) while the other proofs are listed after the corresponding lemmas. Please note that the use of Lemma 35 in the Lemma 46 is therefore not a vicious circle but a valid assumption in the induction step.

Lemma 35 (Entry) *When $wait_j$ is entered, $found_j$ and run_j are false and j is not in any wait set W_i and no process i is going to deliver its result to j:*

$$t_w \rightarrow (\neg found_j \wedge \neg run_j \wedge \forall i(\neg \mathtt{elem}_i(j, W_i) \wedge (\mathtt{at}\{\alpha^i_{12} \ldots \alpha^i_{21}\} \rightarrow w_i \neq j))).$$

Proof: We will prove this lemma by induction over the number of times that $wait_j$ has been executed. Define

(A.1) $B \equiv \neg found_j \wedge \neg run_j \wedge \forall i(\neg \mathtt{elem}_i(j, W_i) \wedge (\mathtt{at}\{\alpha^i_{12} \ldots \alpha^i_{21}\} \rightarrow w_i \neq j))$.

We will prove

(A.2) $\tau_w \rightarrow B$.

Induction Base

We assume that $wait_j$ is entered for the first time i.e.

(A.3) $\tau_\Pi \rightarrow (\tau_w \ \mathtt{atnext}\ \beta^j_1)$.

By $start_\Pi$, we know

(A.4) $\tau_\Pi \rightarrow B$.

Since only $wait_j$ sets $found_j$ to true and extends W_i by j, we therefore know

(A.5) $\tau_w \rightarrow (\neg found_j \wedge \forall i(\neg \mathtt{elem}_i(j, W_i)))$

and also

(A.6) $\tau_\Pi \rightarrow ((\neg found_j \wedge \forall i(\neg \mathtt{elem}_i(j, W_i))) \ \mathtt{until}\ \tau_w)$.

From this, we know

(A.7) $\tau_\Pi \rightarrow (\forall i(\mathtt{at}\{\alpha^i_{12} \ldots \alpha^i_{21}\} \rightarrow w_i \neq j) \ \mathtt{until}\ \tau_w)$

which implies

(A.8) $\tau_w \rightarrow (\neg run_j \wedge \forall i(\mathtt{at}\{\alpha^i_{12} \ldots \alpha^i_{21}\} \rightarrow w_i \neq j))$.

Induction Step

Let τ'_w such that

(A.9) $\tau_w \rightarrow (\tau'_w \ \mathtt{atnext}\ \beta^j_1)$.

From Lemma 46 (Exit), we know

(A.10) $\tau_r \rightarrow B$.

Since only $wait_j$ sets $found_j$ to true and extends W_i by j, it is easy to derive

(A.11) $\tau'_w \rightarrow B$.

Therefore we are done. □

The following lemma establishes a relationship between W_i and R_j.

Lemma 36 (Membership) *If process j is in the wait set W_i, then process i is in the process set $\overline{R_j}$ and j executed $wait_j$:*

$$\tau_w \rightarrow \forall i(\mathtt{elem}_i(j, W_i) \rightarrow i \in \overline{R_j}) \ \mathtt{until}\ \tau_r.$$

Proof: Let i be arbitrary fixed. We know from Lemma 35 (Entry)

(A.1) $\tau_w \rightarrow \neg\mathtt{elem}_i(j, W_i)$.

Now let τ be arbitrary fixed with $\tau_w < \tau < \tau_r$ such that

(A.2) $\tau \rightarrow \mathtt{elem}_i(j, W_i)$.

By the program structure, we know that for some τ_p with $\tau_r < \tau_p < \tau$

(A.3) $\tau_p \rightarrow \beta_7^j \wedge p_j = i$.

By Lemma 28 (Process Set), we then know

(A.4) $\tau_p \rightarrow i \in \overline{R}_j$

and by Lemma 24 (Constancy) we are done. □

The following lemma will be used to guarantee that no Π_i delivers a result to Π_j "too early".

Lemma 37 (Waiting) *If outside the critical region protected by $rsem_j$ j is in some wait set W_i, then process j is ready to receive a result.*

$$\tau_w \rightarrow ((rsem_j \wedge \mathtt{elem}_i(j, W_i)) \rightarrow \mathtt{at}\{\beta_{22}^j, \beta_e^j\})\ \mathtt{until}\ \tau_r$$

Proof: Let $\tau \geq \tau_w$ be arbitrary fixed such that

(A.1) $\tau \rightarrow (rsem_j \wedge \mathtt{elem}_i(j, W_i))$.

By Lemma 35 (Entry), we know

(A.2) $\tau_w \rightarrow \forall i(\neg\mathtt{elem}_i(j, W_i))\ \mathtt{until}\ \tau_d$.

By Lemma 20 (Result Semaphore), we know

(A.3) $\tau_d \rightarrow \neg rsem_j\ \mathtt{until}\ \tau_e$.

We therefore know $\tau > \tau_d$, and thus

(A.4) $\tau_d \rightarrow \mathtt{at}\{\beta_{22}^j \ldots \beta_e^j\}$. □

The next lemma fixes a non-obvious property at the start of the critical region protected by $rsem_j$.

Lemma 38 (Lock) *If i is going to deliver its result to j then j was in the wait set of i when i locked the critical region protected by $rsem_j$:*

$$\tau_\Pi \rightarrow \Box\forall i((\mathtt{at}\{\alpha_{12}^i \ldots \alpha_{21}^i\} \wedge w_i = j) \rightarrow (\mathtt{elem}_i(j, W_i)\ \mathtt{atprev}\ \alpha_7^i)).$$

Proof: Let i be arbitrary fixed and define

(A.1) $D \equiv \forall i((\mathtt{at}\{\alpha_{12}^i \ldots \alpha_{21}^i\} \wedge w_i = j) \rightarrow (\mathtt{elem}_i(j, W_i)\ \mathtt{atprev}\ \alpha_7^i))$.

From $start_\Pi$, we know

(A.2) $\tau_\Pi \rightarrow D$.

Let time $\tau > \tau_\Pi$ be arbitrary fixed such that

(A.3) $\tau \rightarrow (\mathtt{at}\{\alpha_{12}^i \ldots \alpha_{21}^i\} \wedge w_i = j)$.

By Lemma 34 (Deliver), we know

(A.4) $\tau \rightarrow (j \in W_i\ \mathtt{atprev}\ \alpha_9^i)$.

By Lemma 18 (Wait Semaphore), we then know

(A.5) $\tau \to (j \in W_i \ \texttt{atprev}\ \alpha_8^i)$.

By Lemma 21 (Result Exclusion), we then know

(A.6) $\tau \to (\texttt{elem}_i(j, W_i)\ \texttt{atprev}\ \alpha_7^i)$. □

The following lemma guarantees that no Π_i delivers a result to Π_j before Π_j has left the critical region protected by $rsem_j$.

Lemma 39 (Not Run) *After the entry of $wait_j$, run_j remains false until the end of the critical region protected by $rsem_j$:*

$$\tau_w \to (\neg run_j\ \texttt{until}\ \tau_e).$$

Proof: By Lemma 35 (Entry), we know

(A.1) $\tau_w \to \neg run_j$.

By Lemma 35 (Entry) and the program structure, we also know

(A.2) $\tau_w \to (\forall i(\neg \texttt{elem}_i(j, W_i))\ \texttt{until}\ \beta_3^j)$.

By Axiom 11 (Wait Statement), we know

(A.3) $\tau_w \to (\neg rsem_j\ \texttt{atnext}\ \beta_3^j)$.

Moreover, we know by Lemma 38 (Lock) and Axiom 11 (Wait Statement)

(A.4) $\tau_w \to ((rsem_j \wedge \texttt{elem}_i(j, W_i))\ \texttt{before}\ (\alpha_{21}^i \wedge w_i = j))$.

By Lemma 20 (Result Semaphore), (A.2), (A.3) and (A.4) imply

(A.5) $\tau_w \to (\neg run_j\ \texttt{atnext}\ \beta_2^j)$.

By Lemma 21 (Result Exclusion), we are done. □

The next both lemmas describe the invariants that hold at each iteration of the inner respectively outer loop of $wait_j$. Unfortunately, using temporal logic the invariants of nested loops cannot be proved separately since they mutually depend on each other. Therefore, we combine them to one and verify the correctness of the joint proposition

- at the first entry of each loop,
- at the transition to the next iteration of each loop,
- at the exit of each loop,
- at the entry of a loop when another process is executed.

This yields seven cases for both loops (since the exit of the inner loop coincidences with the entry of the outer loop).

Lemma 40 (Remove Loop) *At the start of each iteration of the inner loop of $wait_j$, j is only in the wait sets W_i of all $i \in Q_j$:*

$$\begin{aligned}\tau_w \to ((\ &\texttt{at}\{\beta_{15}^j\} \to \\ &\forall i \in Q_j(\texttt{elem}_i(j, W_i)) \wedge \\ &\forall i \in \overline{R}_j \backslash Q_j(\neg \texttt{elem}_i(j, W_i)))\ \texttt{until}\ \tau_r).\end{aligned}$$

Proof: The proof is contained in the proof of Lemma 41 (Wait Loop). □

Lemma 41 (Wait Loop) *At the start of each iteration of the outer loop of $wait_j$, the following invariant holds:*

1. *If $found_j$ then j is not in any wait set W_i.*
2. *If $\neg found_j$ then j is only in the wait sets W_i of all $i \in \overline{R}_j \backslash P_j$.*

Formally,

$$\begin{aligned}\tau_w \rightarrow ((\ &\mathtt{at}\{\beta_3^j\} \rightarrow \\ &(found_j \wedge \forall i \in \overline{R}_j(\neg\mathtt{elem}_i(j, W_i))) \vee \\ &(\neg found_j \wedge \forall i \in P_j(\neg\mathtt{elem}_i(j, W_i)) \wedge \forall i \in \overline{R}_j \backslash P_j(\mathtt{elem}_i(j, W_i)))\ \mathtt{until}\ \tau_r).\end{aligned}$$

Proof: We will prove this lemma simultaneously with Lemma 40 (Remove Loop). Define

(A.1) $E \equiv found_j \wedge \forall i \in \overline{R}_j(\neg\mathtt{elem}_i(j, W_i))$

(A.2) $F \equiv \neg found_j \wedge \forall i \in P_j(\neg\mathtt{elem}_i(j, W_i)) \wedge \forall i \in \overline{R}_j \backslash P_j(\mathtt{elem}_i(j, W_i))$

(A.3) $G \equiv \forall i \in Q_j(\mathtt{elem}_i(j, W_i)) \wedge \forall i \in \overline{R}_j \backslash Q_j(\neg\mathtt{elem}_i(j, W_i))$

(A.4) $H \equiv (\ \mathtt{at}\{\beta_3^j\} \rightarrow (E \vee F)) \wedge (\ \mathtt{at}\{\beta_{15}^j\} \rightarrow G)$

We will prove

(A.5) $\tau_w \rightarrow (H\ \mathtt{until}\ \tau_r)$.

First we have to prove

(A.6) $\tau_w \rightarrow H$

which is obviously true. Second let τ be arbitrary fixed with $\tau_w < \tau < \tau_r$ and assume

(A.7) $\tau_w \rightarrow (H\ \mathtt{until}\ \tau)$.

We have to prove

(A.8) $\tau \rightarrow H$.

We may restrict our consideration to the following cases

(A.9) $\tau \rightarrow \triangle(\beta_2^j \vee (\ \mathtt{at}\{\beta_3^j\} \wedge \neg\beta_3^j) \vee \beta_9^j \vee \beta_{14}^j) \vee$

(A.10) $\tau \rightarrow \triangle((\ \mathtt{at}\{\beta_{15}^j\} \wedge \neg\beta_{15}^j) \vee (\beta_{15}^j \wedge Q_j = \emptyset) \vee \beta_{20}^j)$

since otherwise $\tau \rightarrow \neg\ \mathtt{at}\{\beta_3^j, \beta_{15}^j\}$.

Case $\tau \rightarrow \triangle\beta_2^j$

By the case assumption, we know

(A.11) $\tau \rightarrow\ \mathtt{at}\{\beta_3^j\}$.

By Lemma 35 (Entry) and Lemma 17 (Locality), we know

(A.12) $\tau \rightarrow (\neg found_j \wedge P_j = \overline{R}_j \wedge \forall i \in \overline{R}_j(\neg\mathtt{elem}_i(j, W_i)))$

and thus

(A.13) $\tau \rightarrow F$.

Case $\tau \rightarrow \triangle(\ \mathtt{at}\{\beta_3^j\} \wedge \neg\beta_3^j)$

By the case assumption, we know

(A.14) $\tau \rightarrow\ \mathtt{at}\{\beta_3^j\}$.

By the induction hypothesis (A.7), we also know

(A.15) $\tau \rightarrow \triangle(E \vee F)$.

By Lemma 17 (Locality) and Lemma 21 (Result Exclusion), we then know

(A.16) $\tau \rightarrow E \vee F$.

Case $\boldsymbol{\tau \rightarrow \triangle\beta_9^j}$

By the case assumption, we know

(A.17) $\tau \rightarrow \mathtt{at}\{\beta_3^j\}$.

By the induction hypothesis (A.7) and Axiom 14 (While Loop), we know

(A.18) $\tau \rightarrow ((H \wedge P_j \neq \emptyset \wedge \neg found_j)\ \mathtt{atprev}\ \beta_3^j)$

which implies

(A.19) $\tau \rightarrow ((F \wedge P_j \neq \emptyset)\ \mathtt{atprev}\ \beta_3^j)$.

By the case assumption, Lemma 21 (Result Exclusion) and Lemma 24 (Constancy), we then can derive with the program structure

(A.20) $\tau \rightarrow F$.

Case $\boldsymbol{\tau \rightarrow \triangle\beta_{14}^j}$

By the case assumption, we know

(A.21) $\tau \rightarrow \mathtt{at}\{\beta_{15}^j\}$.

By the induction hypothesis (A.7) and Axiom 14 (While Loop), we know

(A.22) $\tau \rightarrow ((F \wedge P_j \neq \emptyset)\ \mathtt{atprev}\ \beta_3^j)$.

By Lemma 17 (Locality) and Lemma 21 (Result Exclusion), we can derive

(A.23) $\tau \rightarrow G$.

Case $\boldsymbol{\tau \rightarrow \triangle(\mathtt{at}\{\beta_{15}^j\} \wedge \neg\beta_{15}^j)}$

By the case assumption, we know

(A.24) $\tau \rightarrow \mathtt{at}\{\beta_{15}^j\}$.

By the induction hypothesis (A.7), we know

(A.25) $\tau \rightarrow \triangle G$.

By Lemma 17 (Locality) and Lemma 21 (Result Exclusion), we then know (A.8).

Case $\boldsymbol{\tau \rightarrow \triangle(\beta_{15}^j \wedge Q_j = \emptyset)}$

By the case assumption, we know

(A.26) $\tau \rightarrow \mathtt{at}\{\beta_3^j\}$.

By the program structure and Lemma 17 (Locality), we know

(A.27) $\tau \rightarrow found_j$.

By the induction hypothesis (A.7), we know

(A.28) $\tau \rightarrow \triangle G$.

Lemma 21 (Result Exclusion) therefore implies

(A.29) $\tau \rightarrow E$.

Case $\tau \rightarrow \triangle\beta^{\mathbf{j}}_{\mathbf{20}}$

By the case assumption, we know

(A.30) $\tau \rightarrow \mathtt{at}\{\beta^j_{15}\}$.

By the induction hypothesis (A.7) and Axiom 14 (While Loop), we know

(A.31) $\tau \rightarrow ((G \land Q_j \neq \emptyset)\ \mathtt{atprev}\ \beta^j_5)$.

By Lemma 22 (Set Update) and Lemma 21 (Result Exclusion), we can then derive

(A.32) $\tau \rightarrow G$. □

The following lemma describes the state of Π when the outer loop is left i.e. immediately before the critical region is left.

Lemma 42 (Wait End) *At the end of the main loop of $wait_j$, the process j has either found a result or it is going to be blocked:*

$$\begin{aligned}&\tau_e \rightarrow\\ &\quad \neg run_j \land\\ &\quad (found_j \rightarrow (\forall i(\neg\mathtt{elem}_i(j, W_i)) \land r_j \in \overline{R}_j \land y_r = \overline{x}_{r_j})) \land\\ &\quad (\neg found_j \rightarrow (\forall i \in \overline{R}_j(\mathtt{elem}_i(j, W_i)) \land \forall i \notin \overline{R}_j(\neg\mathtt{elem}_i(j, W_i))))).\end{aligned}$$

Proof: This is a corollary of Lemma 39 (Not Run), Lemma 41 (Wait Loop) and Lemma 21 (Result Exclusion). □

The following two lemmas describe the invariants of the nested loops of $result_i$ and are simultaneously proved similar to the previous invariants. Of particular interest is the invariant of the outer loop which not only describes the state at the entry of the loop but more generally the state of Π when the critical region protected by $rsem_j$ is entered (which happens at the end of each iteration of the outer loop). Therefore we have to check all exits of this critical region in any process.

Lemma 43 (Erase Loop) *At the start of each iteration of the inner loop of $result_i$, w_i is exactly in the wait sets $W_{i'}$ of all $i' \in S_i$:*

$$\begin{aligned}&\tau_w \rightarrow \forall i(((\mathtt{at}\{\alpha^i_{13}\} \land w_i = j) \rightarrow\\ &\quad (\forall i' \in S_i(\mathtt{elem}_{i'}(w_i, W_{i'})) \land \forall i' \in R_{w_i} \backslash S_i(\neg\mathtt{elem}_{i'}(w_i, W_{i'})))) \ \mathtt{until}\ \tau_r).\end{aligned}$$

Proof: The proof is contained in the proof of Lemma 44 (Result Loop). □

Lemma 44 (Result Loop) *If process j is a member of some wait set W_i and no process is in the critical region protected by $rsem_j$, then j is actually a member of all wait sets W_i with $i \in \overline{R}_j$ and j is (at the point of being) blocked:*

$$\begin{aligned}&\tau_w \rightarrow (((rsem_j \land \exists i(\mathtt{elem}_i(j, W_i))) \rightarrow\\ &\quad (\neg found_j \land \neg run_j \land \forall i \in \overline{R}_j(\mathtt{elem}_i(j, W_i)) \land \mathtt{at}\{\beta^j_{22}, \beta^j_{24}\})) \ \mathtt{until}\ \tau_r).\end{aligned}$$

Proof: We prove this lemma simultaneously with Lemma 43 (Erase Loop). Let i be arbitrary fixed and define

(A.1) $X_0 \equiv rsem_j \land \exists i'(\mathtt{elem}_{i'}(j, W_{i'}))$,

(A.2) $X_1 \equiv \neg found_j \land \neg run_j \land \forall i' \in \overline{R}_j(\mathtt{elem}_{i'}(j, W_{i'})) \land \mathtt{at}\{\beta^j_{21}, \beta^j_{22}\}$,

(A.3) $Y_0 \equiv \mathtt{at}\{\alpha^i_{13}\} \land w_i = j$,

(A.4) $Y_1 \equiv \forall i' \in S_i(\mathtt{elem}_{i'}(w_i, W_{i'})) \land \forall i' \in R_{w_i} \backslash S_i(\neg\mathtt{elem}_{i'}(w_i, W_{i'}))$.

We want to prove

(A.5) $\tau_w \rightarrow ((X_0 \rightarrow X_1) \wedge (Y_0 \rightarrow Y_1))$ `until` τ_r.

We know by Lemma 35 (Entry)

(A.6) $\tau_w \rightarrow (\neg X_0 \wedge \neg Y_0)$

Let time τ be arbitrary fixed with $\tau_w < \tau < \tau_r$ and assume

(A.7) $\tau_w \rightarrow (((X_0 \rightarrow X_1) \wedge (Y_0 \rightarrow Y_1))$ `until` τ).

We have to prove

(A.8) $\tau \rightarrow ((X_0 \rightarrow X_1) \wedge (Y_0 \rightarrow Y_1))$.

We may restrict our considerations to the following cases:

1. $\boldsymbol{\tau \rightarrow \triangle\beta^{j}_{21}}$: the critical region protected by $rsem_j$ is left and X_0 may become true.
2. $\boldsymbol{\tau \rightarrow \triangle(\beta^{j}_{22} \vee \beta^{j}_{24})}$: $\mathtt{at}\{\beta^j_{22}, \beta^j_{24}\}$ may become false.
3. $\boldsymbol{\tau \rightarrow \triangle(w_i = j \wedge (\alpha^{i}_{12} \vee \mathtt{at}\{\alpha^{i}_{13}\} \vee \alpha^{i}_{18}))}$: Y_0 becomes true.
4. $\boldsymbol{\tau \rightarrow \triangle(\alpha^{i'}_{22} \wedge w_{i'} = j)}$: the critical region protected by $rsem_j$ is left and X_0 may become true.

For all other actions, obviously either

- $\tau \rightarrow \neg X_0 \wedge \neg Y_0$ i.e. the premises are false or
- $\tau \rightarrow (\triangle X_1 \rightarrow X_1)$ and $\tau \rightarrow (\triangle Y_1 \rightarrow Y_1)$ i.e. the conclusions are not changed.

Case $\boldsymbol{\tau \rightarrow \triangle\beta^{j}_{21}}$

We assume

(A.9) $\tau \rightarrow X_0$.

We then know by the case assumption and Lemma 42 (Wait End)

(A.10) $\tau \rightarrow \triangle(\neg run_j \wedge found_j \wedge \forall i \in \overline{R}_j(\mathtt{elem}_i(j, W_i)))$

and thus by the program structure also

(A.11) $\tau \rightarrow (\neg run_j \wedge found_j \wedge \forall i \in \overline{R}_j(\mathtt{elem}_i(j, W_i)) \wedge \mathtt{at}\{\beta^j_{22}\})$.

Therefore, we know

(A.12) $\tau \rightarrow X_1$.

By Lemma 21 (Result Exclusion), we also know

(A.13) $\tau \rightarrow \neg Y_0$.

Case $\boldsymbol{\tau \rightarrow \triangle\beta^{j}_{22}}$

We assume

(A.14) $\tau \rightarrow X_0$.

The case assumption then implies

(A.15) $\tau \rightarrow \triangle X_0$

and thus by the induction hypothesis (A.7)

(A.16) $\tau \rightarrow \triangle X_1$.

This implies

(A.17) $\tau \rightarrow \triangle \neg found_j$

and thus by Axiom 13 (One Branch Conditional)

(A.18) $\tau \rightarrow \mathtt{at}\{\beta^j_{24}\}$.

From this and (A.16), we know

(A.19) $\tau \rightarrow X_1$.

If we assume

(A.20) $\tau \rightarrow Y_0$

we also know by the program structure

(A.21) $\tau \rightarrow \triangle Y_0$

and thus by the induction hypothesis (A.7) and the program structure

(A.22) $\tau \rightarrow Y_1$.

Case $\boldsymbol{\tau \rightarrow \triangle \beta^j_{24}}$

The case assumption implies by Axiom 11 (Wait Statement)

(A.23) $\tau \rightarrow \triangle run_j$.

Then we know by the induction hypothesis (A.7)

(A.24) $\tau \rightarrow \triangle \neg X_0$

and by the program structure also

(A.25) $\tau \rightarrow \neg X_0$.

Let us assume

(A.26) $\tau \rightarrow Y_0$.

Then we know by the case assumption

(A.27) $\tau \rightarrow \triangle Y_0$

and thus by the induction hypothesis (A.7) and the program structure

(A.28) $\tau \rightarrow Y_1$.

Case $\boldsymbol{\tau \rightarrow \triangle(\alpha^i_{12} \wedge w_i = j)}$

By Axiom 11 (Wait Statement) and Lemma 38 (Lock), we know

(A.29) $\tau \rightarrow ((rsem_j \wedge \mathtt{elem}_i(j, W_i))\ \mathtt{atprev}\ \alpha^i_7)$.

By the induction hypothesis (A.7), we thus know

(A.30) $\tau \rightarrow (X_0\ \mathtt{atprev}\ \alpha^i_7)$.

By the program structure, Lemma 21 (Result Exclusion) and Lemma 18 (Wait Semaphore), we then know

(A.31) $\tau \rightarrow \forall i' \in \overline{R}_j(\mathtt{elem}_{i'}(j, W_{i'}))$

and thus also

(A.32) $\tau \rightarrow Y_1$.

By Lemma 21 (Result Exclusion), we also know

(A.33) $\tau \rightarrow \neg X_0$.

Case $\tau \rightarrow \triangle(\texttt{at}\{\alpha^{\mathbf{i}}_{\mathbf{13}}\} \wedge \mathbf{w_i} = \mathbf{j})$

By the induction hypothesis (A.7) we have

(A.34) $\tau \rightarrow \triangle Y_1$

and thus by Lemma 21 (Result Exclusion)

(A.35) $\tau \rightarrow Y_1$.

By Lemma 21 (Result Exclusion), we also know

(A.36) $\tau \rightarrow \neg X$.

Case $\tau \rightarrow \triangle(\alpha^{\mathbf{i}}_{\mathbf{18}} \wedge \mathbf{w_i} = \mathbf{j})$

By the induction hypothesis (A.7) and Axiom 14 (While Loop), we know

(A.37) $\tau \rightarrow ((Y_1 \wedge S_i \neq \emptyset)$ `atprev` $\alpha^i_{13})$.

ant then by Lemma 21 (Result Exclusion) and Lemma 18 (Wait Semaphore)

(A.38) $\tau \rightarrow (\forall i' \in S_i(\texttt{elem}_{i'}(j, W_{i'})) \wedge \forall i' \in \overline{R}_j \backslash S_i(\neg\texttt{elem}_{i'}(j, W_{i'})))$.

By Lemma 21 (Result Exclusion), we then have

(A.39) $\tau \rightarrow Y_1$.

By Lemma 21 (Result Exclusion), we also know

(A.40) $\tau \rightarrow \neg X_0$.

Case $\tau \rightarrow \triangle(\alpha^{\mathbf{i'}}_{\mathbf{23}} \wedge \mathbf{w_{i'}} = \mathbf{j})$

We know by Lemma 21 (Result Exclusion)

(A.41) $\tau \rightarrow \neg Y_0$.

We know by Axiom 11 (Wait Statement)

(A.42) $\tau \rightarrow (rsem_j$ `atprev` $\alpha^{i'}_7))$.

We will now distinguish two cases:

1. **Case $\tau \rightarrow (\mathbf{j} \in \mathbf{W_{i'}}$ `atprev` $\alpha^{\mathbf{i'}}_{\mathbf{7}})$**

   We will show

   (A.43) $\tau \rightarrow \neg X_0$

   by proving

   (A.44) $\tau \rightarrow \forall k(\neg\texttt{elem}_k(j, W_k))$.

   By the induction hypothesis (A.7), we know

   (A.45) $\tau \rightarrow ((\texttt{elem}_{i'}(j, W_{i'}) \wedge \forall k \in R_j(\texttt{elem}_k(j, W_k)))$ `atprev` $\alpha^{i'}_7)$.

   Then we also have, by Lemma 21 (Result Exclusion),

   (A.46) $\tau \rightarrow (j \in W_{i'}$ `atprev` $\alpha^{i'}_9)$.

From this and the program structure, we then know

(A.47) $\tau \rightarrow ((\bigcirc\, \mathtt{at}\{\alpha_{12}^{i'}\})\ \mathtt{atprev}\ \alpha_{11}^{i'})$.

Furthermore, by the induction hypothesis (A.7) we know

(A.48) $\tau \rightarrow (\forall k(\neg\mathtt{elem}_k(j, W_k))\ \mathtt{atprev}\ \alpha_{19}^{i'})$

which leads by Lemma 21 (Result Exclusion) to (A.44).

2. **Case $\boldsymbol{\tau \rightarrow (\mathbf{j} \notin \mathbf{W_{i'}}\ \mathtt{atprev}\ \alpha_7^{i'})}$**

We assume

(A.49) $\tau \rightarrow X_0$.

By Lemma 21 (Result Exclusion), we know

(A.50) $\tau \rightarrow (w_{i'} \notin W_{i'}\ \mathtt{atprev}\ \alpha_9^{i'})$

and thus by Axiom 13 (One Branch Conditional)

(A.51) $\tau \rightarrow ((\bigcirc\, \mathtt{at}\{\alpha_{22}^{i'}\})\ \mathtt{atprev}\ \alpha_{11}^{i'})$

From this we know by (A.49) and Lemma 21 (Result Exclusion)

(A.52) $\tau \rightarrow (X_0\ \mathtt{atprev}\ \alpha_7^{i'})$.

From the induction hypothesis (A.7), we thus know

(A.53) $\tau \rightarrow (X_1\ \mathtt{atprev}\ \alpha_7^{i'})$

Then we have by (A.50), (A.51) and Lemma 21 (Result Exclusion)

(A.54) $\tau \rightarrow X_1$. □

The final case distinction in the previous proof actually describes an interesting situation. It may happen that some Π_i with $i \in R_j$ finds at α_5 that $j \in W_i$ but has to detect at α_9 that $j \notin W_i$! In this case, some other $\Pi_{i'}$ with $i' \in R_j$ has executed α_5 simultaneously to Π_i but was slightly faster in executing α_7. It "stole" therefore j from W_i and delivered $x_{i'}$ to j. The lock of $rsem_j$ at α_7 is therefore necessary to determine a unique winner of the "race".

Lemma 45 (Result End) *If process i is going to awake process j, j is not in any wait set $W_{i'}$ any more:*

$$\tau_w \rightarrow \Box\forall i((\alpha_{21}^i \wedge w_i = j) \rightarrow \forall i'(\neg\mathtt{elem}_{i'}(j, W_{i'})))$$

Proof: Let i be arbitrary fixed and let $\tau > \tau_w$ such that

(A.1) $\tau \rightarrow (\alpha_{21}^i \wedge w_i = j)$.

By the program structure, we then know

(A.2) $\tau \rightarrow (S_i = \emptyset\ \mathtt{atprev}\ \alpha_{13}^i)$.

By Lemma 43 (Erase Loop), this implies

(A.3) $\tau \rightarrow (\forall i'(\neg\mathtt{elem}_{i'}(j, W_{i'})\ \mathtt{atprev}\ \alpha_{13}^i)$.

By Lemma 21 (Result Exclusion), we therefore know

(A.4) $\tau \rightarrow \forall i'(\neg\mathtt{elem}_{i'}(j, W_{i'}))$. □

The last lemma in this section describes the state of Π immediately before $wait_j$ is left. This lemma is necessary for two purposes:

1. To guarantee that Lemma 35 (Entry) holds for the next execution of $wait_j$.
2. To guarantee that, after some Π_i has set the values of r_j and y_j, they are not changed any more by any $\Pi_{i'}$.

Lemma 46 (Exit) *When $wait_j$ is left, $found_j$ and run_j are false and j is not in any wait set W_i and no process i is going to deliver its result to j:*

$$t_f \rightarrow ((\neg found_j \wedge \neg run_j \wedge \forall i(\neg \mathtt{elem}_i(j, W_i) \wedge (\mathtt{at}\{\alpha_1 2^i \dots \alpha^i_{21}\} \rightarrow w_i \neq j))) \\ \mathtt{until}\ \tau_r)$$

Proof: By Lemma 39 (Not Run), we know

(A.1) $\tau_e \rightarrow \neg run_j$.

We will now distinguish two cases.

1. **Case $\boldsymbol{\tau_e \rightarrow}$ found$_\mathbf{j}$**

   By the program structure and Lemma 17 (Locality), we know

   (A.2) $\tau_e \rightarrow ((\neg found_j)\ \mathtt{until}\ \tau_r)$.

   By Lemma 20 (Result Semaphore), we know

   (A.3) $\tau_e \rightarrow \forall i(\ \mathtt{at}\{\alpha^i_8 \dots \alpha^i_{22}\} \rightarrow w_i \neq j)$.

   By (A.1), Lemma 42 (Wait End) and the program structure, we know

   (A.4) $\tau_e \rightarrow (\forall i(\neg \mathtt{elem}_i(j, W_i))\ \mathtt{until}\ \tau_r)$

   and with (A.3) therefore

   (A.5) $\tau_e \rightarrow (\forall i((\alpha^i_9 \wedge w_i = j) \rightarrow (w_i \notin W_j))\ \mathtt{until}\ \tau_r)$

   By (A.1) and the program structure, it is then easy to derive

   (A.6) $\tau_e \rightarrow (B\ \mathtt{upto}\ \tau_r)$.

2. **Case $\boldsymbol{\tau_e \rightarrow \neg}$found$_\mathbf{j}$**

   By the program structure, we know

   (A.7) $\tau_e \rightarrow (\beta^j_{24}\ \mathtt{before}\ \tau_r)$.

   Define τ_g with $\tau_e < \tau_g < \tau_r$ such that

   (A.8) $\tau_e \rightarrow (\tau_g\ \mathtt{atnext}\ \beta^j_{24})$.

   By Axiom 11 (Wait Statement), we know

   (A.9) $\tau_g \rightarrow run_j$.

   By (A.1) this implies for some i

   (A.10) $\tau_e \rightarrow ((\alpha^i_{21} \wedge w_i = j)\ \mathtt{before}\ \tau_g)$.

   Therefore, we know by the program structure and Lemma 45 (Result End)

   (A.11) $\tau_e \rightarrow (\forall i(\neg \mathtt{elem}_i(j, W_i))\ \mathtt{before}\ \tau_g)$.

   From this and the definition of α^i_{21}, we can derive

   (A.12) $\tau_f \rightarrow (B\ \mathtt{upto}\ \tau_r)$

   and are done. □

A.5 Continuation Lemmas

In this section, we deal with all lemmas expressing eventuality properties. In our case, these are assertions that

1. Critical regions protected by a semaphore are left.
2. While loops terminate.

The organization of proofs is as follows:

1. **Wait Continuation:** Lemma 47 says that the critical regions protected by $wsem_i$ are left. These regions do not contain other critical regions or while loops.
2. **Remove Induction:** Lemma 48 says that the inner loop of $wait_j$ (which only contains $wsem_i$ regions) terminates.
3. **Wait Induction:** Lemma 49 says that the outer loop of $wait_j$ (which contains the $wsem_i$ region and an inner loop) terminates.
4. **Erase Induction:** Lemma 50 says that the inner loop of $result_i$ (which contains a critical region protected by $wsem_i$) terminates.
5. **Result Continuation:** Lemma 51 says that the critical regions protected by $rsem_j$ are left. These regions contain the outer loop of $wait_j$ and the inner loop of $result_i$.
6. **Result Induction:** Lemma 54 shows that the outer loop of $result_i$ (which contains critical regions protected by $rsem_j$ and $wsem_i$ and an inner loop) terminates.
7. **First Out:** Lemma 55 provides an alternative condition that is sufficient to prove Theorem 5 (Sometimes).

Lemma 47 (Wait Continuation) *No critical region protected by some $wsem_i$ will be occupied forever:*

$$\tau_\Pi \rightarrow \Box \forall i (\Diamond wsem_i).$$

Proof: Let time $\tau \geq \tau_\Pi$ and i be arbitrary fixed and assume

(A.1) $\tau \rightarrow \neg wsem_i$.

By Lemma 18 (Wait Semaphore), we then know

(A.2) $\tau \rightarrow \mathtt{at}\{\alpha_4^i \ldots \alpha_6^i, \alpha_9^i, \alpha_{10}^i, \alpha_{24}^i\} \vee$
(A.3) $\tau \rightarrow \exists i'(s_{i'} = i \wedge \mathtt{at}\{\alpha_{16}^{i'}, \alpha_{17}^{i'}\}) \vee$
(A.4) $\tau \rightarrow \exists j(p_j = i \wedge \mathtt{at}\{\beta_6^j \ldots \beta_8^j, \beta_{10}^j\}) \vee$
(A.5) $\tau \rightarrow \exists j(q_j = i \wedge \mathtt{at}\{\beta_{18}^j, \beta_{19}^j\})$.

All of the denoted regions are terminated by $\mathbf{V}(wsem_i)$ and no one contains an **endwhile** or a **P** statement. Hence, we know by Axiom 16 (Fairness)

(A.6) $\tau \rightarrow \Diamond wsem_i$. $\Box$

Lemma 48 (Remove Induction) *The inner loop in $wait_j$ terminates:*

$$\tau_\Pi \rightarrow \Box(\beta_{15}^j \rightarrow \Diamond(\beta_{15}^j \wedge Q_j = \emptyset)).$$

Proof: This can be easily verified by induction over Q_j i.e. by proving the assertion

$$\begin{aligned}&\tau_\Pi \to \Box\forall\overline{Q}(\\&\quad(\beta^j_{15} \wedge Q_j = \overline{Q} \wedge \overline{Q} \neq \emptyset) \to\\&\qquad((\bigcirc\Diamond\beta^j_{15}) \wedge (Q_j \subset \overline{Q}\ \texttt{atnext}\ \beta^j_{15})))\end{aligned}$$

by the application of Lemma 47 (Wait Continuation) and Axiom 16 (Fairness). □

Lemma 49 (Wait Induction) *The outer loop in $wait_j$ terminates:*

$$\tau_\Pi \to \Box(\beta^j_3 \to \Diamond(\beta^j_3 \wedge (P_j = \emptyset \vee found_j)).$$

Proof: We will prove this by induction over P_j i.e. by proving the assertion

$$\begin{aligned}&\tau_\Pi \to \Box\forall\overline{P}(\\&\quad(\beta^j_3 \wedge P_j = \overline{P} \wedge \overline{P} \neq \emptyset \wedge \neg found_j) \to\\&\qquad((\bigcirc\Diamond\beta^j_3) \wedge ((found_j \vee P_j \subset \overline{P})\ \texttt{atnext}\ \beta^j_3))).\end{aligned}$$

Let time $\tau > \tau_\Pi$ and $\overline{P}$ be arbitrary fixed such that

(A.1) $\tau \to (\beta^j_3 \wedge P_j = \overline{P} \wedge \overline{P} \neq \emptyset)$.

By Axiom 16 (Fairness), it is clear that

(A.2) $\tau \to \bigcirc\Diamond\beta^j_6$.

We will distinguish two cases:

1. Case $\boldsymbol{\tau \to (\neg \mathbf{avail}_{\mathbf{p_j}}}\ \texttt{atnext}\ \boldsymbol{\beta^j_6))}$

   By Lemma 47 (Wait Continuation) and Lemma 16 (Fairness), we know

   (A.3) $\tau \to \bigcirc\Diamond\beta^j_3$.

   Furthermore, we can easily show

   (A.4) $\tau \to (P_j \subset \overline{P}\ \texttt{atnext}\ \beta^j_3)$.

2. Case $\boldsymbol{\tau \to (\mathbf{avail}_{\mathbf{p_j}}}\ \texttt{atnext}\ \boldsymbol{\beta^j_5))}$

   By Lemma 48 (Remove Induction), we can prove

   (A.5) $\tau \to \bigcirc\Diamond\beta^j_3$.

   Furthermore, by the program structure it is easy to show

   (A.6) $\tau \to (found_j\ \texttt{atnext}\ \beta^j_3)$. □

Lemma 50 (Erase Induction) *The inner loop in $result_i$ terminates:*

$$\tau_\Pi \to \Box\forall i(\alpha^i_{13} \to \Diamond(\alpha^i_{13} \wedge S_i = \emptyset)).$$

Proof: This can be easily verified by induction over S_i i.e. by proving the assertion

$$\begin{aligned}&\tau_\Pi \to \Box\forall i\forall\overline{S}(\\&\quad(\alpha^i_{16} \wedge S_i = \overline{S} \wedge \overline{S} \neq \emptyset) \to\\&\qquad((\bigcirc\Diamond\alpha^i_{16}) \wedge (S_i \subset \overline{S}\ \texttt{atnext}\ \alpha^i_{16})))\end{aligned}$$

by the application of Lemma 47 (Wait Continuation) and Axiom 16 (Fairness). □

Lemma 51 (Result Continuation) *No critical region protected by some $rsem_j$ will be occupied forever:*

$$\tau_\Pi \to \Box\Diamond rsem_j.$$

Proof: Let time $\tau \geq \tau_\Pi$ be arbitrary fixed such that

(A.1) $\tau \rightarrow \neg rsem_j$.

We will prove

(A.2) $\tau \rightarrow \Diamond rsem_j$.

By Lemma 20 (Result Semaphore), we have one the following cases:

1. **Case $\boldsymbol{\tau \rightarrow \mathtt{at}\{\beta_3^j \ldots \beta_{21}^j\}}$**

   By Lemma 49 (Wait Induction), we can prove

   (A.3) $\tau \rightarrow \Diamond(\beta_3^j \wedge (P_j = \emptyset \vee found_j))$.

   By Axiom 16 (Fairness), we then know

   (A.4) $\tau \rightarrow \Diamond \beta_{20}^j$

   and are done.

2. **Case $\boldsymbol{\tau \rightarrow (w_i = j \wedge \mathtt{at}\{\alpha_8^i \ldots \alpha_{22}^i\})}$**

   By Lemma 47 (Wait Continuation), we know

   (A.5) $\tau \rightarrow \Diamond \alpha_9^i$.

   We may assume

   (A.6) $\tau \rightarrow (j \in W_i \ \mathtt{atnext}\ \alpha_9^i)$

   since otherwise Axiom 13 (One Branch Conditional) and Axiom 16 (Fairness) would directly lead to (A.8). Then we know by the program structure and Lemma 50 (Erase Induction)

   (A.7) $\tau \rightarrow \Diamond(\alpha_{13}^i \wedge S_i = \emptyset)$

   which leads to

   (A.8) $\tau \rightarrow \Diamond \alpha_{22}^i$. □

The following lemma says that $result_j$ puts j only in such W_i where the corresponding Π_i has not yet terminated.

Lemma 52 (Terminate) *If process j has put the identifier j into some wait set W_i, then the process i has not yet terminated:*

$$\tau_w \rightarrow ((\mathtt{at}\{\beta_3^j\} \wedge \neg found_j) \rightarrow \forall i \in \overline{R}_j \backslash P_j (\neg\, \mathtt{at}\{\alpha_{24}^i, \alpha_e^i\}))\ \mathtt{until}\ \tau_r.$$

Proof: Define

(A.1) $D \equiv \forall i \in \overline{R}_j \backslash P_j (\neg\, \mathtt{at}\{\alpha_{24}^j, \alpha_e^j\}))$.

By the definition of τ_w, we know

(A.2) $\tau_w \rightarrow \neg\, \mathtt{at}\{\beta_3^j\}$.

Let time τ be arbitrary fixed with $\tau_w < \tau < \tau_r$ and assume

(A.3) $\tau \rightarrow (\mathtt{at}\{\beta_3^j\} \wedge \neg found_j)$.

We are going to prove

(A.4) $\tau \rightarrow D$.

We know by the program structure

(A.5) $\tau \to \triangle((\mathtt{at}\{\beta_3^j\} \wedge \neg\beta_3^j) \vee \beta_2^j \vee \beta_9^j \vee \beta_{15}^j)$

and may therefore restrict our considerations to the following cases.

Case $\tau \to \triangle(\mathtt{at}\{\beta_3^j\} \wedge \neg\beta_3^j)$

By the induction hypothesis, we know

(A.6) $\tau \to \triangle\forall i \in \overline{R}_j\backslash P_j(\neg\,\mathtt{at}\{\alpha_{23}^i, \alpha_e^i\})$.

By (A.3) and Lemma 41 (Wait Loop), we also know

(A.7) $\tau \to \triangle\forall i \in \overline{R}_j\backslash P_j(\mathtt{elem}_i(j, P_i))$.

and therefore

(A.8) $\tau \to \triangle\forall i \in \overline{R}_j(\alpha_4^i \to W_i \neq \emptyset)$

and are thus done.

Case $\tau \to \triangle\beta_2^j$

By the program structure and Lemma 17 (Locality), we know

(A.9) $\tau \to P_j = \overline{R}_j$

and are therefore done.

Case $\tau \to \triangle\beta_{15}^j$

By the program structure and Lemma 30 (Found), we know

(A.10) $\tau \to found_j$

and are therefore done.

Case $\tau \to \triangle\beta_9^j$

Let i be arbitrary fixed such that

(A.11) $\tau \to i \in \overline{R}_j\backslash P_j$.

Let time $\tau_1 > \tau_w$ such that

(A.12) $\tau \to (\tau_1\ \mathtt{atprev}\ \beta_3^j)$.

We will distinguish two cases:

1. **Case $\tau_1 \to i \in \overline{R}_j\backslash P_j$**

   By the induction hypothesis, we know

   (A.13) $\tau_1 \to \neg\,\mathtt{at}\{\alpha_{24}^i, \alpha_e^i\}$.

   Lemma 41 (Wait Loop) and (A.3) imply

   (A.14) $\tau_1 \to \mathtt{elem}_i(j, W_i)$

   By Lemma 20 (Result Semaphore), we also know

   (A.15) $\tau_1 \to (\neg rsem_j\ \mathtt{until}\ \tau)$

   and thus by Lemma 21 (Result Exclusion)

   (A.16) $\tau_1 \to (\mathtt{elem}_i(j, W_i)\ \mathtt{until}\ \tau)$.

   But then we also know

   (A.17) $\tau_1 \to ((\alpha_4^i \to W_i \neq \emptyset)\ \mathtt{until}\ \tau)$

   and are therefore done.

2. **Case $\tau_1 \rightarrow i \in P_j$**

   Let time τ_2 such that

   (A.18) $\tau \rightarrow (\tau_2 \texttt{ atprev } \beta_6^j)$.

   We know by the case assumption and Axiom 12 (Two Branch Conditional)

   (A.19) $\tau_2 \rightarrow (p_j = i \wedge \neg avail_i)$

   and therefore by Lemma 25 (Available)

   (A.20) $\tau_2 \rightarrow \neg \texttt{at}\{\alpha_3^i \ldots \alpha_e^i\}$.

   Let τ_3 such that

   (A.21) $\tau_2 \rightarrow (\tau_3 \texttt{ atnext } \beta_8^j)$.

   From Lemma 18 (Wait Semaphore) and (A.20), we can conclude

   (A.22) $\tau_3 \rightarrow \neg \texttt{at}\{\alpha_4^i \ldots \alpha_e^i\}$.

   Furthermore, we know by the program structure and Lemma 21 (Result Exclusion)

   (A.23) $\tau_3 \rightarrow \texttt{elem}_i(j, W_i)$.

   From both assertions we can conclude

   (A.24) $\tau_3 \rightarrow ((\alpha_4^i \rightarrow j \in W_i) \texttt{ until } \tau)$

   and therefore

   (A.25) $\tau_3 \rightarrow (\neg \texttt{at}\{\alpha_{24}^i, \alpha_e^i\} \texttt{ until } \tau)$. □

Lemma 53 (Iteration) *If j is in W_i outside the critical region protected by $rsem_j$, then eventually a new iteration of the outer loop of $result_j$ will start.*

$$\tau_w \rightarrow (\forall i((rsem_j \wedge \texttt{elem}_i(j, W_i)) \rightarrow \Diamond\alpha_4^i \texttt{ until } \tau_r).$$

Proof: Let τ with $\tau_r \leq \tau < \tau_w$ and i be arbitrary fixed such that

(A.1) $\tau \rightarrow rsem_j \wedge \texttt{elem}_i(j, W_i)$.

By Lemma 37 (Waiting), we know

(A.2) $\tau_w \rightarrow (\beta_{21}^j \texttt{ before } \tau)$.

By the program structure, there exists a time τ' with $\tau_w < \tau' < \tau$ such that

(A.3) $\tau' \rightarrow (\beta_3^j \wedge \neg found_j \wedge P_j = \emptyset)$.

By Lemma 52 (Terminate) and Lemma 36 (Membership), we then know

(A.4) $\tau' \rightarrow \neg \texttt{at}\{\alpha_{24}^i, \alpha_e^i\}$

i.e. we know by Lemma 43 (Erase Loop) and Lemma 51 (Result Continuation)

(A.5) $\tau' \rightarrow \Diamond\alpha_4^i$

and are done. □

The following lemma shows that not only $wait_j$ but also $result_i$ terminates. This is the main constituent in the proof of Theorem 5 (Sometime).

Lemma 54 (Result Induction) *The outer loop in $result_i$ terminates:*

$$\tau_w \rightarrow \forall i((rsem_j \wedge \texttt{elem}_i(j, W_i)) \rightarrow \Diamond(\alpha_4^i \wedge W_i = \emptyset)) \texttt{ until } \tau_r.$$

Proof: Let τ with $\tau_w < \tau < \tau_r$ and i be arbitrary fixed such that

(A.1) $\tau \rightarrow (rsem_j \wedge \mathtt{elem}_i(j, W_i))$.

By Lemma 53 (Iteration), we know

(A.2) $\tau \rightarrow \Diamond\alpha_4^i$.

Let us assume that for some $\overline{W} \neq \emptyset$

(A.3) $\tau' \rightarrow W_i = \overline{W}$.

It now suffices to show

(A.4) $\tau' \rightarrow (W_i \subset \overline{W}\ \mathtt{atnext}\ \alpha_4^i)$.

By Lemma 25 (Available), we know

(A.5) $\tau' \rightarrow \Box avail_i$.

This implies by Lemma 18 (Wait Semaphore)

(A.6) $\tau' \rightarrow \Box\forall j(\beta_7^j \rightarrow p_j \neq i)$

i.e. W_i will be not extended any more by any other process. Define j such that

(A.7) $\tau' \rightarrow (w_i = j\ \mathtt{atnext}\ \alpha_6^i)$.

We know by Lemma 18 (Wait Semaphore)

(A.8) $\tau' \rightarrow j \in W_i$.

In case of

(A.9) $\tau' \rightarrow (j \notin W_i\ \mathtt{atnext}\ \alpha_9^i)$

we are done. Otherwise, Lemma 45 (Result End) implies

(A.10) $\tau' \rightarrow (j \notin W_i\ \mathtt{atnext}\ \alpha_{21}^i)$

and we are done. $\Box$

The following lemma can be proved using the additional Axiom 18 (Queue) and may be used in an alternative proof of Theorem 5 (Sometime) in case that (by a modification in the program structure) Lemma 54 (Result Induction) does not hold:

Lemma 55 (First Out) *No j will remain in W_i forever:*

$$\tau_w \rightarrow \forall i((rsem_j \wedge \mathtt{elem}_i(j, W_i)) \rightarrow \Diamond(\neg\mathtt{elem}_i(j, W_i)))\ \mathtt{until}\ \tau_r.$$

Proof: Let τ with $\tau_w < \tau < \tau_r$ and i be arbitrary fixed such that

(A.1) $\tau \rightarrow (rsem_j \wedge \mathtt{elem}_i(j, W_i))$.

By Lemma 53 (Iteration), we know that for some time $\tau' > \tau$

(A.2) $\tau' \rightarrow \alpha_4^i$.

We may assume

(A.3) $\tau' \rightarrow \mathtt{elem}_i(j, W_i)$

since otherwise we are done. By Lemma 51 (Result Continuation) and Lemma 43 (Erase Loop), we know

(A.4) $\tau' \rightarrow \Box(\alpha_4^i \rightarrow (W_i = \emptyset \vee \bigcirc\Diamond\alpha_4^i))$

i.e. the iteration will continue forever unless W_i becomes empty. We may assume

(A.5) $\tau' \rightarrow \Box(\alpha_4^i \rightarrow \bigcirc\Diamond\alpha_4^i)$

since otherwise we are done (see Lemma 54 (Result Induction)). By the program structure and Lemma 43 (Erase Loop), we know for some j'

(A.6) $\tau' \rightarrow \bigcirc((j' = w_i \wedge w_i \notin W_i)$ `atnext` $\alpha_4^i)$.

We may assume $j \neq j'$ since otherwise we are done. But, by Axiom 18 (Queue), we know

(A.7) $\tau' \rightarrow j' \in W_i$.

Since there are only finitely many such j', we know with (A.5)

(A.8) $\tau' \rightarrow \Diamond w_i = j$

and are therefore done. □

Appendix B

The pD Language

Sections B.1 and B.2 give the concrete syntax of pD. Section B.3 describes the interface to the runtime system.

B.1 Terminal Symbols

The syntax of the terminal symbols is described in Unix 'lex' (actually GNU flex) syntax.

```
/*****************************************************************************
 *
 * pD terminal symbols
 *
 *****************************************************************************/

letter [a-zA-Z]
digit  [0-9]
letdig [a-zA-Z0-9]
strch  [^\"]

ident  {letter}{letdig}*
number {digit}{digit}*
string \"{strch}*\"

%%

 /* end of file */
<<EOF>> return(EOFOP);

 /* whitespace */
\n
\t
[ ]
--[^\n]*
"{-"

 /* preprocessor token */
#

 /* keywords */
"let"     return(LETOP);
"where"   return(WHEREOP);
"in"      return(INOP);
```

```
"if"      return(IFOP);
"then"    return(THENOP);
"else"    return(ELSEOP);
"case"    return(CASEOP);
"of"      return(OFOP);
"type"    return(TYPEOP);
"loop"    return(LOOPOP);
"list"    return(LISTOP);
"comp"    return(COMPOP);
"rlist"   return(RLISTOP);
"rcomp"   return(RCOMPOP);
"ndcomp"  return(NDCOMPOP);
"extern"  return(EXTERNOP);
"proc"    return(PROCOP);
"const"   return(CONSTOP);
"start"   return(STARTOP);
"wait"    return(WAITOP);
"any"     return(ANYOP);
"stop"    return(STOPOP);
"scons"   return(SCONSOP);
"snil"    return(SNILOP);
"stream"  return(STREAMOP);
"para"    return(PARAOP);
"scase"   return(SCASEOP);
"_null"   return(NULLOP);
"_head"   return(HEADOP);
"_tail"   return(TAILOP);

 /* special symbols */
"="    return(DEFOP);
"->"   return(FUNARROP);
"=>"   return(CASEARROP);
":"    return(CONSOP);
"::"   return(SIGOP);
";"    return(SEMICOLOP);
","    return(COMMAOP);
"."    return(DOTOP);
"|"    return(BAROP);
"<-"   return(GENARROP);
".."   return(DOTSOP);
"+="   return(BYOP);
"-="   return(DOWNBYOP);
"@"    return(ATOP);
"!@"   return(ATEXCLAMOP);
"|@"   return(BARATOP);
"'"    return(PRIMEOP);
":@"   return(CONSATOP);

 /* parentheses */
"("    return(LPAROP);
")"    return(RPAROP);
"{"    return(LBRACEOP);
"}"    return(RBRACEOP);
"["    return(LBRACKOP);
"]"    return(RBRACKOP);

 /* arithmetic operators */
"+"    return(ADDOP);
"-"    return(SUBOP);
"*"    return(MULOP);
"/"    return(DIVOP);
"%"    return(MODOP);
"&&"   return(ANDOP);
"||"   return(OROP);
"^"    return(NOTOP);
```

```
 /* comparison operators */
"=="   return(EQOP);
"/="   return(NEQOP);
"<"    return(LTOP);
"<="   return(LEQOP);
">"    return(GTOP);
">="   return(GEQOP);

 /* reserved words */
"True"   return(TRUEOP);
"False"  return(FALSEOP);
"Nil"    return(NILOP);
"Int"    return(INTOP);
"Bool"   return(BOOLOP);
"Word"   return(WORDOP);
"Error"  return(ERROROP);
"Task"   return(TASKOP);
"Stream" return(PSTREAMOP);
"Tuple"  return(TUPLEOP);
"List"   return(LLISTOP);

 /* terminal classes */
{number} { return(NUMBER); }
{ident}  { return(IDENT); }
{string} { return(STRING); }
```

B.2 Context-Free Grammar

The context-free grammar is described in Unix 'yacc' (actually GNU bison) notation.

```
/******************************************************************************
 *
 * pD context-free grammar
 *
 ******************************************************************************/

/* keywords */
%token LETOP IFOP THENOP ELSEOP CASEOP OFOP TYPEOP INOP
%token LOOPOP LISTOP COMPOP RLISTOP RCOMPOP EXTERNOP WHEREOP PROCOP CONSTOP
%token TASKOP STARTOP WAITOP ANYOP STOPOP NDCOMPOP SCONSOP SNILOP
%token STREAMOP RSTREAMOP PARAOP SSTREAMOP PSTREAMOP TUPLEOP LLISTOP
%token HEADOP TAILOP NULLOP SCASEOP

/* special symbols */
%token DEFOP FUNARROP CASEARROP CONSOP SIGOP SEMICOLOP COMMAOP DOTOP
%token BAROP GENARROP DOTSOP BYOP DOWNBYOP ATOP ATEXCLAMOP BARATOP PRIMEOP
%token CONSATOP

/* parentheses */
%token LPAROP RPAROP LBRACEOP RBRACEOP LBRACKOP RBRACKOP

/* arithmetic operators */
%token ADDOP SUBOP MULOP DIVOP MODOP ANDOP OROP NOTOP

/* comparison operators */
%token EQOP NEQOP LTOP LEQOP GTOP GEQOP

/* predefined identifiers */
%token TRUEOP FALSEOP NILOP INTOP BOOLOP WORDOP ERROROP

/* terminal classes */
```

```
%token <tokval> NUMBER IDENT STRING

/* end of file */
%token EOFOP

%nonassoc LETOP WHEREOP
%right    OROP
%right    ANDOP
%nonassoc EQOP NEQOP LTOP LEQOP GTOP GEQOP
%right    CONSOP CONSATOP
%left     ADDOP SUBOP
%left     MULOP DIVOP MODOP
%nonassoc NOTOP
%left     DOTOP
%left     ATOP
%left     ATEXCLAMOP
%left     FUNARROP

/* program */
program:
  expression EOFOP
;

/* statement sequence */
statseq:
  statement semicols statseq
|
;

/* semicolons */
semicols:
  semicols SEMICOLOP
|
;

/* statement */
statement:
  IDENT SIGOP type
| EXTERNOP IDENT SIGOP type
| EXTERNOP PROCOP IDENT SIGOP type
| EXTERNOP CONSTOP IDENT SIGOP type
| PARAOP IDENT SIGOP type
| EXTERNOP TYPEOP IDENT
| TYPEOP IDENT DEFOP type
| TYPEOP IDENT LPAROP tidentlist RPAROP DEFOP type
| IDENT DEFOP expression
| IDENT LPAROP patternlist RPAROP DEFOP expression
| pattern DEFOP expression
| error
| EOFOP
;

/* simple pattern */
pattern:
  LPAROP identlist RPAROP
;

/* comma separated pattern/ident list */
patternlist:
  pattern COMMAOP patternlist
| IDENT COMMAOP patternlist
| pattern
| IDENT
;
```

```
/* type identifier list */
tidentlist:
  MULOP IDENT COMMAOP tidentlist
| MULOP IDENT
;

/* identifier list */
identlist:
   IDENT COMMAOP identlist
|  IDENT
;

/* type expressions */
type:
  INTOP
| BOOLOP
| WORDOP
| LBRACKOP type RBRACKOP
| LLISTOP LPAROP type RPAROP
| LPAROP typelist RPAROP
| TUPLEOP LPAROP typelist RPAROP
| LPAROP RPAROP FUNARROP type
| type FUNARROP type
| IDENT
| IDENT LPAROP typelist RPAROP
| MULOP IDENT
| TASKOP LPAROP type RPAROP
| SSTREAMOP LPAROP type RPAROP
| PSTREAMOP LPAROP type RPAROP
;

/* list of types */
typelist:
  type COMMAOP typelist
| type
;

/* expression */
expression:
  blockexp
| condexp
| lcaseexp
| loopexp
| parexp
| basicexp
| listexp
;

/* local bindings */
blockexp:
  LETOP LBRACEOP statseq RBRACEOP INOP expression
| expression WHEREOP LBRACEOP statseq RBRACEOP
;

/* conditional expressions */
condexp:
  IFOP expression THENOP expression ELSEOP expression
;

/* list case expressions */
lcaseexp:
  CASEOP expression OFOP LBRACEOP
  niltag CASEARROP expression semicols
  IDENT CONSOP IDENT CASEARROP expression semicols
  RBRACEOP
```

```
;

/* stream case expressions */
scaseexp:
  SCASEOP expression OFOP LBRACEOP
  niltag CASEARROP expression semicols
  IDENT CONSOP IDENT CASEARROP expression semicols
  RBRACEOP
;

/* nil tag for case expression */
niltag:
  NILOP
| LBRACKOP RBRACKOP
;

/* loop expression */
loopexp:
  composer LBRACKOP expression BAROP looplist RBRACKOP
;

composer:
  LOOPOP LPAROP expression RPAROP
| LISTOP
| RLISTOP
| STREAMOP
| COMPOP LPAROP expression COMMAOP expression RPAROP
| RCOMPOP LPAROP expression COMMAOP expression RPAROP
| NDCOMPOP LPAROP expression COMMAOP expression RPAROP
| NDCOMPOP LPAROP expression COMMAOP expression COMMAOP expression RPAROP
;

/* loop list */
looplist:
  loop BAROP   looplist
| loop BARATOP expression looplist
| loop BARATOP looplist
| loop
|
;

/* loop */
loop:
  genlist
| SEMICOLOP expression
| SEMICOLOP SEMICOLOP nextseq
| genlist SEMICOLOP expression
| genlist SEMICOLOP SEMICOLOP nextseq
| SEMICOLOP expression SEMICOLOP nextseq
| genlist SEMICOLOP expression SEMICOLOP nextseq
;

/* next definition */
nextstat:
  statement
| IDENT PRIMEOP DEFOP expression
;

/* next sequence separated by commas */
nextseq:
  nextstat COMMAOP nextseq
| nextstat
|
;
```

```
/* generator list */
genlist:
  generator COMMAOP genlist
| generator
|
;

/* iteration generator */
generator:
  IDENT GENARROP expression DOTSOP expression BYOP expression
| IDENT GENARROP expression DOTSOP expression DOWNBYOP expression
| IDENT GENARROP expression DOTSOP expression
| IDENT GENARROP expression BYOP expression
| IDENT GENARROP expression DOWNBYOP expression
| IDENT GENARROP expression DOTSOP
| IDENT GENARROP expression
| IDENT CONSOP IDENT GENARROP expression
| statement
;

/* parallel expressions */
parexp:
  STARTOP LPAROP IDENT COMMAOP explist RPAROP
| WAITOP LPAROP expression RPAROP
| ANYOP LPAROP expression RPAROP
| STOPOP expression INOP expression
| expression ATOP expression
| expression ATOP
| expression ATEXCLAMOP
| expression CONSATOP expression
| SCONSOP LPAROP expression COMMAOP expression RPAROP
| SNILOP
| HEADOP LPAROP expression RPAROP
| TAILOP LPAROP expression RPAROP
| NULLOP LPAROP expression RPAROP
| scaseexp
;

/* basic expressions */
basicexp:
  NOTOP expression
| SUBOP expression
| expression EQOP  expression
| expression NEQOP expression
| expression LTOP  expression
| expression LEQOP expression
| expression GTOP  expression
| expression GEQOP expression
| expression ADDOP expression
| expression SUBOP expression
| expression MULOP expression
| expression DIVOP expression
| expression MODOP expression
| expression ANDOP expression
| expression OROP  expression
| expression CONSOP expression
| NILOP
| TRUEOP
| FALSEOP
| NUMBER
| IDENT
| LPAROP explist RPAROP
| IDENT LPAROP explist RPAROP
| IDENT LPAROP RPAROP
| ERROROP LPAROP STRING RPAROP
```

```
;

/* list expressions */
listexp:
  LBRACKOP RBRACKOP
| LBRACKOP listexp0 RBRACKOP
;

listexp0:
  expression COMMAOP listexp0
| expression
;

/* expression list */
explist:
  expression COMMAOP explist
| expression
;
```

B.3 The Runtime Interface

```
/******************************************************************************
 *
 * dc.h
 * runtime system interface of pD compiler
 *
 * Copyright (C) 1993 Wolfgang Schreiner, see file COPYRIGHT.
 *
 ******************************************************************************/

#ifndef DC_H
#define DC_H

#if defined(DC_NOINLINE) || !defined(__GNUC__)
#define __inline__
#endif

#ifdef DC_UNIX

/******************************************************************************
 * simple Unix-based runtime system
 ******************************************************************************/

#include <stdlib.h>
#include <stdio.h>

#define WORD  int

#define INT   WORD
#define BOOL  WORD
#define POLY  WORD
#define NILT  WORD
#define SGNAL WORD

#define FALSE 0
#define TRUE  1
#define NIL   0

#define ERROR(s) do {fprintf(stderr, s); abort();} while (0)

#define CONS(h,t)  cons(h,t)
#define RCONS(h,t) rcons(h,t)
```

```
#define ISNIL(l)   (l == NIL)
#define HEAD(l)    (((WORD*)l)[0])
#define TAIL(l)    (((WORD*)l)[1])

#define HEAPSIZ 1000000

WORD _heap_[HEAPSIZ];
WORD *_heapptr_ = _heap_;
WORD *_heapend_ = _heap_+HEAPSIZ;

__inline__
static WORD *new(void) {
  _heapptr_ += 2;
  if (_heapptr_ == _heapend_) ERROR("Heap overflow\n");
  return ((WORD*)_heapptr_);
}

__inline__
static WORD cons(WORD h, WORD t) {
  WORD *m = new();
  m[0] = h;
  m[1] = t;
  return((WORD)m);
}

__inline__
static WORD* rcons(WORD h, WORD *t) {
  WORD *m = new();
  *(WORD**)t = m;
  m[0] = h;
  return(m+1);
}

#define TUPLE2(a,b) \
  CONS(a, CONS(b, NIL))
#define TUPLE3(a,b,c) \
  CONS(a, CONS(b, CONS(c, NIL)))
#define TUPLE4(a,b,c,d) \
  CONS(a, CONS(b, CONS(c, CONS(d, NIL))))
#define TUPLE5(a,b,c,d,e) \
  CONS(a, CONS(b, CONS(c, CONS(d, CONS(e, NIL)))))
#define TUPLE6(a,b,c,d,e,f) \
  CONS(a, CONS(b, CONS(c, CONS(d, CONS(e, CONS(f, NIL))))))

#define ELEM1(l) \
  HEAD(l)
#define ELEM2(l) \
  HEAD(TAIL(l))
#define ELEM3(l) \
  HEAD(TAIL(TAIL(l)))
#define ELEM4(l) \
  HEAD(TAIL(TAIL(TAIL(l))))
#define ELEM5(l) \
  HEAD(TAIL(TAIL(TAIL(TAIL(l)))))
#define ELEM6(l) \
  HEAD(TAIL(TAIL(TAIL(TAIL(TAIL(l))))))

#elif defined(DC_SACLIB)

/****************************************************************************
 * SACLIB runtime system
 ****************************************************************************/

#include "saclib.h"
#include <stdio.h>
```

```
#define WORD  Word

#define INT   WORD
#define BOOL  WORD
#define POLY  WORD
#define NILT  WORD
#define SGNAL WORD

#define FALSE 0
#define TRUE  1

#define ERROR(s) do {fprintf(stderr, s); abort();} while (0)

__inline__
static Word _COMP(Word a, Word L) {
  WORD M;
  if (AVAIL == NIL) GC();
  M = AVAIL;
  AVAIL = RED(M);
  SFIRST(M,a);
  SRED(M,L);
  return(M);
}

__inline__
static Word *_RCOMP(Word a, Word *L) {
  WORD M;
  if (AVAIL == NIL) GC();
  M = AVAIL;
  AVAIL = RED(M);
  SFIRST(M,a);
  *L = M;
  return(SPACEB+M);
}

#define CONS(h,t)  _COMP(h,t)
#define RCONS(h,t) _RCOMP(h,t)
#define HEAD(l)    FIRST(l)
#define TAIL(l)    RED(l)

#define TUPLE2(a,b) \
  CONS(a, CONS(b, NIL))
#define TUPLE3(a,b,c) \
  CONS(a, CONS(b, CONS(c, NIL)))
#define TUPLE4(a,b,c,d) \
  CONS(a, CONS(b, CONS(c, CONS(d, NIL))))
#define TUPLE5(a,b,c,d,e) \
  CONS(a, CONS(b, CONS(c, CONS(d, CONS(e, NIL)))))
#define TUPLE6(a,b,c,d,e,f) \
  CONS(a, CONS(b, CONS(c, CONS(d, CONS(e, CONS(f, NIL))))))
#define TUPLE(l) l

#define ELEM1(l) \
  HEAD(l)
#define ELEM2(l) \
  HEAD(TAIL(l))
#define ELEM3(l) \
  HEAD(TAIL(TAIL(l)))
#define ELEM4(l) \
  HEAD(TAIL(TAIL(TAIL(l))))
#define ELEM5(l) \
  HEAD(TAIL(TAIL(TAIL(TAIL(l)))))
#define ELEM6(l) \
  HEAD(TAIL(TAIL(TAIL(TAIL(TAIL(l))))))
```

```
#define ELEMN(n,l) LELTI(l,n)

#define START0(f) f()
#define START1(f,a) f(a)
#define START2(f,a,b) f(a,b)
#define START3(f,a,b,c) f(a,b,c)
#define START4(f,a,b,c,d) f(a,b,c,d)
#define START5(f,a,b,c,d,e) f(a,b,c,d,e)
#define START6(f,a,b,c,d,e,g) f(a,b,c,d,e,g)
#define START7(f,a,b,c,d,e,g,h) f(a,b,c,d,e,g,h)
#define START8(f,a,b,c,d,e,g,h,i) f(a,b,c,d,e,g,h,i)
#define START9(f,a,b,c,d,e,g,h,i,j) f(a,b,c,d,e,g,h,i,j)
#define START10(f,a,b,c,d,e,g,h,i,j,k) f(a,b,c,d,e,g,h,i,j,k)
#define START11(f,a,b,c,d,e,g,h,i,j,k,l) f(a,b,c,d,e,g,h,i,j,k,l)
#define START12(f,a,b,c,d,e,g,h,i,j,k,l,m) f(a,b,c,d,e,g,h,i,j,k,l,m)
#define START13(f,a,b,c,d,e,g,h,i,j,k,l,m,n) f(a,b,c,d,e,g,h,i,j,k,l,m,n)
#define START14(f,a,b,c,d,e,g,h,i,j,k,l,m,n,o) \
  f(a,b,c,d,e,g,h,i,j,k,l,m,n,o)
#define START15(f,a,b,c,d,e,g,h,i,j,k,l,m,n,o,p) \
  f(a,b,c,d,e,g,h,i,j,k,l,m,n,o,p)
#define START16(f,a,b,c,d,e,g,h,i,j,k,l,m,n,o,p,q) \
  f(a,b,c,d,e,g,h,i,j,k,l,m,n,o,p,q)

#define FSTART0(r,f) (r+f())
#define FSTART1(r,f,a) (r+f(a))
#define FSTART2(r,f,a,b) (r+f(a,b))
#define FSTART3(r,f,a,b,c) (r+f(a,b,c))
#define FSTART4(r,f,a,b,c,d) (r+f(a,b,c,d))
#define FSTART5(r,f,a,b,c,d,e) (r+f(a,b,c,d,e))
#define FSTART6(r,f,a,b,c,d,e,g) (r+f(a,b,c,d,e,g))
#define FSTART7(r,f,a,b,c,d,e,g,h) (r+f(a,b,c,d,e,g,h))
#define FSTART8(r,f,a,b,c,d,e,g,h,i) (r+f(a,b,c,d,e,g,h,i))

#define FORWARD() (0)

#define WAIT(t) (t)
#define ANY(t)  (ISNIL(t) ? TUPLE2(0,0) : TUPLE2(HEAD(t),TAIL(t)))
#define STOP(t) do { } while(0)

#else

/******************************************************************************
 * PACLIB runtime system
 ******************************************************************************/

#include "paclib.h"

#ifdef DC_REAL
#define pacCreate pacStart
#define pacCreateFwd pacStartFwd
#else
#define pacCreate pacVirtual
#define pacCreateFwd pacVirtualFwd
#endif

#define WORD  Word

#define INT   WORD
#define BOOL  WORD
#define POLY  WORD
#define NILT  WORD
#define SGNAL WORD
#define TASK  WORD

#define FALSE 0
```

```
#define TRUE  1

#define ERROR(s) do { pacAbort("D2C Error %s\n", s); } while (0)

#define CONS(h,t)  COMP(h,t)
#define RCONS(h,t) RCOMP(h,t)
#define HEAD(l)    FIRST(l)
#define TAIL(l)    RED(l)

#define TUPLE2(a,b) \
  CONS(a, CONS(b, NIL))
#define TUPLE3(a,b,c) \
  CONS(a, CONS(b, CONS(c, NIL)))
#define TUPLE4(a,b,c,d) \
  CONS(a, CONS(b, CONS(c, CONS(d, NIL))))
#define TUPLE5(a,b,c,d,e) \
  CONS(a, CONS(b, CONS(c, CONS(d, CONS(e, NIL)))))
#define TUPLE6(a,b,c,d,e,f) \
  CONS(a, CONS(b, CONS(c, CONS(d, CONS(e, CONS(f, NIL))))))
#define TUPLE(l) l

#define ELEM1(l) \
  HEAD(l)
#define ELEM2(l) \
  HEAD(TAIL(l))
#define ELEM3(l) \
  HEAD(TAIL(TAIL(l)))
#define ELEM4(l) \
  HEAD(TAIL(TAIL(TAIL(l))))
#define ELEM5(l) \
  HEAD(TAIL(TAIL(TAIL(TAIL(l)))))
#define ELEM6(l) \
  HEAD(TAIL(TAIL(TAIL(TAIL(TAIL(l))))))
#define ELEMN(n,l) LELTI(l,n)

#define START0(f) pacCreate(f,0)
#define START1(f,a) pacCreate(f,1,a)
#define START2(f,a,b) pacCreate(f,2,a,b)
#define START3(f,a,b,c) pacCreate(f,3,a,b,c)
#define START4(f,a,b,c,d) pacCreate(f,4,a,b,c,d)
#define START5(f,a,b,c,d,e) pacCreate(f,5,a,b,c,d,e)
#define START6(f,a,b,c,d,e,g) pacCreate(f,6,a,b,c,d,e,g)
#define START7(f,a,b,c,d,e,g,h) pacCreate(f,7,a,b,c,d,e,g,h)
#define START8(f,a,b,c,d,e,g,h,i) pacCreate(f,8,a,b,c,d,e,g,h,i)
#define START9(f,a,b,c,d,e,g,h,i,j) pacCreate(f,9,a,b,c,d,e,g,h,i,j)
#define START10(f,a,b,c,d,e,g,h,i,j,k) pacCreate(f,10,a,b,c,d,e,g,h,i,j,k)
#define START11(f,a,b,c,d,e,g,h,i,j,k,l) pacCreate(f,11,a,b,c,d,e,g,h,i,j,k,l)
#define START12(f,a,b,c,d,e,g,h,i,j,k,l,m) \
  pacCreate(f,12,a,b,c,d,e,g,h,i,j,k,l,m)
#define START13(f,a,b,c,d,e,g,h,i,j,k,l,m,n) \
  pacCreate(f,13,a,b,c,d,e,g,h,i,j,k,l,m,n)
#define START14(f,a,b,c,d,e,g,h,i,j,k,l,m,n,o) \
  pacCreate(f,14,a,b,c,d,e,g,h,i,j,k,l,m,n,o)
#define START15(f,a,b,c,d,e,g,h,i,j,k,l,m,n,o,p) \
  pacCreate(f,15,a,b,c,d,e,g,h,i,j,k,l,m,n,o,p)
#define START16(f,a,b,c,d,e,g,h,i,j,k,l,m,n,o,p,q) \
  pacCreate(f,16,a,b,c,d,e,g,h,i,j,k,l,m,n,o,p,q)

#define FSTART0(r,f) pacCreateFwd(r,f,0)
#define FSTART1(r,f,a) pacCreateFwd(r,f,1,a)
#define FSTART2(r,f,a,b) pacCreateFwd(r,f,2,a,b)
#define FSTART3(r,f,a,b,c) pacCreateFwd(r,f,3,a,b,c)
#define FSTART4(r,f,a,b,c,d) pacCreateFwd(r,f,4,a,b,c,d)
#define FSTART5(r,f,a,b,c,d,e) pacCreateFwd(r,f,5,a,b,c,d,e)
#define FSTART6(r,f,a,b,c,d,e,g) pacCreateFwd(r,f,6,a,b,c,d,e,g)
```

```
#define FSTART7(r,f,a,b,c,d,e,g,h) pacCreateFwd(r,f,7,a,b,c,d,e,g,h)
#define FSTART8(r,f,a,b,c,d,e,g,h,i) pacCreateFwd(r,f,8,a,b,c,d,e,g,h,i)

#define FORWARD() pacForward()

#define WAIT(t) pacWaitOne(t)
#define ANY(t)  ({WORD t0, t1; t0 = pacWaitList(&t1, t); TUPLE2(t0, t1)})
#define STOP(t) pacStopList(t)

#endif

/*****************************************************************************
 * SACLIB extensions
 *****************************************************************************/
#include "dcsac.h"

#endif

/*****************************************************************************
 * end of dc.h
 *****************************************************************************/

/*****************************************************************************
 *
 * sacext.h
 * saclib extensions of runtime system
 *
 * Copyright (C) 1993 Wolfgang Schreiner, see file COPYRIGHT.
 *
 *****************************************************************************/

#ifndef SACEXT_H
#define SACEXT_H

/* pseudo integer-digit conversion */
__inline__ static WORD DTOI(WORD a) {
  return(a);
}
__inline__ static WORD ITOD(WORD a) {
  return(a);
}

/* other pseudo conversion functions */
__inline__ static WORD DTOIP(WORD a) {
  return(a);
}
__inline__ static WORD ITOIP(WORD a) {
  return(a);
}
__inline__ static WORD IPTOI(WORD a) {
  return(a);
}
__inline__ static WORD DTOMP(WORD a) {
  return(a);
}
__inline__ static WORD MPTOD(WORD a) {
  return(a);
}
__inline__ static WORD MPTOIP(WORD a) {
  return(a);
}

/* polymorphic polynomial functions */
__inline__ static WORD MPDEG(WORD a) {
  return(PDEG(a));
```

```
}
__inline__ static WORD IPDEG(WORD a) {
  return(PDEG(a));
}
__inline__ static WORD MPINV(WORD a, WORD b, WORD c) {
  return(PINV(a, b, c));
}
__inline__ static WORD IPINV(WORD a, WORD b, WORD c) {
  return(PINV(a, b, c));
}
__inline__ static WORD IPDEGSV(WORD a, WORD b, WORD c) {
  return(PDEGSV(a, b, c));
}
__inline__ static WORD MPDEGSV(WORD a, WORD b, WORD c) {
  return(PDEGSV(a, b, c));
}
__inline__ static WORD IPFBRE(WORD a, WORD b) {
  return(PFBRE(a, b));
}
__inline__ static WORD MPFBRE(WORD a, WORD b) {
  return(PFBRE(a, b));
}
__inline__ static WORD IPEQUAL(WORD a, WORD b) {
  return(EQUAL(a, b));
}
__inline__ static WORD MPEQUAL(WORD a, WORD b) {
  return(EQUAL(a, b));
}

/* some constant polynomials */
__inline__ static WORD PCONST2(WORD a, WORD b) {
  return(CONS(a, CONS(b, NIL)));
}
#define IPCONST2(a, b) PCONST2(a, b)
#define MPCONST2(a, b) PCONST2(a, b)

__inline__ static WORD PCONST4(WORD a, WORD b, WORD c, WORD d) {
  return(CONS(a, CONS(b, CONS(c, CONS(d, NIL)))));
}
#define IPCONST4(a, b, c, d) PCONST4(a, b, c, d)
#define MPCONST4(a, b, c, d) PCONST4(a, b, c, d)

/* polynomial to monomial list */
__inline__ static WORD PMonoList(WORD P) {

  if (P == 0)
    return(NIL);
  else {
    WORD L;
    WORD *LP = &L;
    while (!ISNIL(P)) {
      WORD A, B;
      A = FIRST(P);
      P = RED(P);
      B = FIRST(P);
      P = RED(P);
      LP = RCONS(CONS(A, CONS(B, NIL)), LP);
    }
    *LP = NIL;
    return(L);
  }

}

#define IPMonoList(P) PMonoList(P)
```

```
#define MPMonoList(P) PMonoList(P)

/* monomial list to monomial standard representation */
__inline__ static WORD PStdRep(WORD L) {

  if (ISNIL(L))
    return(0);
  else {
    WORD P;
    WORD *PP = &P;
    do {
      WORD T = FIRST(L);
      WORD A = FIRST(T);
      WORD B = FIRST(RED(T));
      if (B) {
        PP = RCONS(A, PP);
        PP = RCONS(B, PP);
      }
      L = RED(L);
    } while(!ISNIL(L));
    *PP = NIL;
    return(P);
  }

#define IPStdRep(L) PStdRep(L)
#define MPStdRep(L) PStdRep(L)

}

#endif

/*****************************************************************************
 * end of sacext.h
 *****************************************************************************/
```

Appendix C

Target Code

In this appendix we present the source and the target code of the linear equation solver presented in Section 4.6 and parallelized in Section 8.3.

Section C.1 gives the functional D program that also shows the interaction between D and SACLIB C. Section C.2 contains the corresponding para-functional solution in pD.

C.1 Sequential Linear Equation Solver

The program consists of several files:

1. The `Makefile` for the program.
2. `linSolve.d` contains the core algorithm.
3. `linCheck.d` checks the correctness of the solution.
4. `linRand.d` generates a random equation system
5. `linDets.c` computes the modular determinants.
6. `linMain.c` contains the input/output interface.

Before compilation, a D source file is automatically passed through the GNU C preprocessor that resolves all `#include` references by including library functions and external signatures.

For each D program the C code generated by the compiler is listed. The algorithm for determinant computation iteratively updates of a two-dimensional array and thus has been implemented directly in C. The generation of a random system has been implemented in D applying the SACLIB random generator, which is a non-functional but nevertheless comfortable "hack".

C.1.1 `Makefile`

```
################################################################################
#
# Makefile for linMain
#
# Compiler Usage: d2c [options] file
# Options: -a (use SACLIB runtime system)
```

```
#         -c (apply conditional transformation)
#         -e (emit external prototypes)
#         -h (help, print this message)
#         -i (inline list constructor functions)
#         -m mainid (name external main function)
#         -n (do not inline functions)
#         -p (print expression before code generation)
#         -q (print expression after each transformation)
#         -r (print addresses of syntax tree nodes)
#         -s (print symbol table information)
#         -t (pointer arguments from program result tuple)
#         -u (use standalone UNIX runtime system)
#         -v (verbose, print compilation statistics)
#         -y (use real tasks instead of virtual tasks)
#         -z (check for sharing)
#
###############################################################################

# saclib root directory
SACLIB = /usr/local/saclib/saclib1.1

# dc library directory, must also set environment variable DCLIBS to "-I ..."
DCLIB = /usr/local/paclib/dc/lib

# compilation
CC = gcc
CFLAGS = -O2 -I ${SACLIB}/include -I ${DCLIB}

all: linmain

linsolve.c: linsolve.d
        d2c -e -m linSolve linsolve.d

lincheck.c: lincheck.d
        d2c -e -m linCheck lincheck.d

linrand.c: linrand.d
        d2c -t -e -m linRand linrand.d

linmain: linsolve.o linrand.o lincheck.o lindets.o main.o
        ${CC} linsolve.o linrand.o lincheck.o lindets.o main.o \
        ${SACLIB}/lib/saclibo.a -o linmain
```

C.1.2 linSolve.d

```
-------------------------------------------------------------------------------
--
-- x = linSolve(A, b, n)
--
-- Solve a linear equation system ("depth-first" version)
--
-- A: an Integer matrix of dimension n that is regular
-- b: an Integer vector of length n
-- n: a Digit
--
-- x: the unique Rational-Vector of length n such that Ax = b
--
-------------------------------------------------------------------------------
linSolve(A, b, n) where {

  -- program arguments
  extern A :: IntMatrix
```

```
  extern b :: IntVector
  extern n :: Digit

  -- libraries
  #include "dclib.i"
  #include "saclib.i"

  -- type names
  type IntVector = [Integer]
  type IntMatrix = [IntVector]
  type RatVector = [Rational]

  -- solve Ax=b for x
  linSolve :: (IntMatrix, IntVector, Digit) -> RatVector
  linSolve(A, b, n) = x where

    B = bound(A, b, n)
    base = (DTOI(1), list [ DTOI(1) | i <- 1..n ])

    (d, ys) = rcomp(cra, base) [ (d, y, p, P) |
        p <- PRIME, P = DTOI(1)
      ; ICOMP(P, B) == -1
      ; P' = if d /= 0 then IPROD(DTOI(p), P) else P,
        (d, y) = linDets(A, b, n, p)
     |; d /= 0 ]

    x = list [ RNRED(y, d) | y <- ys ]

  -- prime bound: P = 1+n!*m^n where m = max(Aij, bi)
  bound :: (IntMatrix, IntVector, Digit) -> Integer
  bound(A, b, n) = P where
    m0 = rcomp(IMAX, DTOI(0)) [ IABSF(x) | x <- b ]
    m1 = rcomp(IMAX, m0) [ IABSF(x) | l <- A | x <- l ]
    P = IPROD(DTOI(2), IPROD(IFACTL(n), IEXP(m1, n)))

  -- modular computation of determinants
  extern proc
  linDets :: (IntMatrix, IntVector, Digit, Digit) -> (Digit, DigitList)

  -- Chinese remainder algorithm
  type IIV = (Integer, IntVector)
  cra :: ((Digit, DigitList, Digit, Integer), IIV) -> IIV
  cra((d, ys, p, P), (D0, Y0)) = (D1, Y1) where
    i = MDINV(p, MDHOM(p, P))
    D1 = MIDCRA(P, p, i, D0, d)
    Y1 = list[ MIDCRA(P, p, i, Y, y) | y <- ys, Y <- Y0 ]

}
------------------------------------------------------------------------------
-- end of linSolve.d
------------------------------------------------------------------------------

/*
 * Generated by DC
 */

#define DC_SACLIB
#undef DC_NOINLINE
#undef DC_UNIX
#undef DC_REAL

#include "dc.h"

/*
 * type definitions
```

```
 */

typedef WORD LIST_347;
typedef WORD TUPLE_348;
... omitted ...
typedef WORD TUPLE_554;

/*
 * external functions and constants
 */

Integer IDP2(Integer, INT);
Integer MIPROD(Integer, Integer, Integer);
... omitted ...
Integer IORD2(INT);

/*
 * local function prototypes
 */

static inline Integer loop_616(IntegerList_97, Integer);
static inline Integer loop_627(IntegerList_97, Integer);
static inline Integer loop_622(IntMatrix_100, Integer);
static inline IntegerList_97 loop_629(INT, INT);
static inline IntegerList_97 loop_610(LIST_347, IntegerList_97, INT, Integer,
  INT);
static inline IIV_104 loop_635(LIST_347, Integer, Integer, IntegerList_97,
  IntMatrix_100, IntegerList_97, INT, Integer);
static inline RatVector_101 loop_643(IntegerList_97, Integer);

/*
 * local function prototypes
 */

static inline Integer loop_616(IntegerList_97 xl_613, Integer _615)
{
  while (!ISNIL(xl_613)) {
    _615 = IMAX(IABSF(HEAD(xl_613)), _615);
    xl_613 = TAIL(xl_613);
  }
  return (_615);
}

static inline Integer loop_627(IntegerList_97 xl_624, Integer _626)
{
  while (!ISNIL(xl_624)) {
    _626 = IMAX(IABSF(HEAD(xl_624)), _626);
    xl_624 = TAIL(xl_624);
  }
  return (_626);
}

static inline Integer loop_622(IntMatrix_100 ll_619, Integer _621)
{
  while (!ISNIL(ll_619)) {
    _621 = loop_627(HEAD(ll_619), _621);
    ll_619 = TAIL(ll_619);
  }
  return (_621);
}

static inline void loop_842(INT i_598, INT n_647, IntegerList_97 * _843)
{
  while (i_598 <= n_647) {
    _843 = RCONS(DTOI(1), _843);
```

```
    i_598 = (i_598 + 1);
  }
  *_843 = NIL;
  return;
}

static inline IntegerList_97 loop_629(INT i_598, INT n_647)
{
  IntegerList_97 _843;
  loop_842(i_598, n_647, &_843);
  return (_843);
}

static inline void loop_844(LIST_347 yl_605, IntegerList_97 Yl_606, INT p_658,
  Integer P_659, INT i_660, IntegerList_97 * _845)
{
  while (!ISNIL(yl_605))
    if (ISNIL(Yl_606)) {
      *_845 = NIL;
      return;
    } else {
      _845 = RCONS(MIDCRA(P_659, p_658, i_660, HEAD(Yl_606), HEAD(yl_605)),
        _845);
      Yl_606 = TAIL(Yl_606);
      yl_605 = TAIL(yl_605);
    }
  *_845 = NIL;
  return;
}

static inline IntegerList_97 loop_610(LIST_347 yl_605, IntegerList_97 Yl_606,
  INT p_658, Integer P_659, INT i_660)
{
  IntegerList_97 _845;
  loop_844(yl_605, Yl_606, p_658, P_659, i_660, &_845);
  return (_845);
}

static inline void loop_846(LIST_347 pl_632, Integer P_600, Integer _637,
  IntegerList_97 _638, IntMatrix_100 A_649, IntegerList_97 b_650, INT n_651,
  Integer B_652, Integer * _847, IntegerList_97 * _848)
{
  while (!ISNIL(pl_632))
    if (ICOMP(P_600, B_652) == -1) {
      INT _849;
      LIST_347 _850;
      INT p_827 = HEAD(pl_632);
      linDets(A_649, b_650, n_651, p_827, &_849, &_850);
      if (_849 != 0) {
        INT i_830 = MDINV(p_827, MDHOM(p_827, P_600));
        _638 = loop_610(_850, _638, p_827, P_600, i_830);
        _637 = MIDCRA(P_600, p_827, i_830, _637, _849);
        P_600 = IPROD(DTOI(p_827), P_600);
        pl_632 = TAIL(pl_632);
      } else
        pl_632 = TAIL(pl_632);
    } else {
      *_847 = _637;
      *_848 = _638;
      return;
    }
  *_847 = _637;
  *_848 = _638;
  return;
}
```

```
static inline IIV_104 loop_635(LIST_347 pl_632, Integer P_600, Integer _637,
  IntegerList_97 _638, IntMatrix_100 A_649, IntegerList_97 b_650, INT n_651,
  Integer B_652)
{
  IntegerList_97 _847;
  Integer _848;
  loop_846(pl_632, P_600, _637, _638, A_649, b_650, n_651, B_652, &_847, &_848);
  return (TUPLE2(_847, _848));
}

static inline void loop_851(IntegerList_97 yl_640, Integer d_645,
  RatVector_101 * _852)
{
  while (!ISNIL(yl_640)) {
    _852 = RCONS(RNRED(HEAD(yl_640), d_645), _852);
    yl_640 = TAIL(yl_640);
  }
  *_852 = NIL;
  return;
}

static inline RatVector_101 loop_643(IntegerList_97 yl_640, Integer d_645)
{
  RatVector_101 _852;
  loop_851(yl_640, d_645, &_852);
  return (_852);
}

/*
 * external entry function
 */

RatVector_101 linSolve(IntMatrix_100 A, IntegerList_97 b, INT n)
{
  Integer _853;
  IntegerList_97 _854;
  loop_846(PRIME, DTOI(1), DTOI(1), loop_629(1, n), A, b, n, IPROD(DTOI(2),
    IPROD(IFACTL(n), IEXP(loop_622(A, loop_616(b, DTOI(0))), n))), &_853,
    &_854);
  return (loop_643(_854, _853));
}

/*
 * End of file
 */
```

C.1.3 linCheck.d

```
--------------------------------------------------------------------------------
--
-- r = linCheck(A, x, b)
--
-- Check the solution of a linear equation system
--
-- A: an Integer Matrix
-- x: a Rational Vector
-- b: an Integer Vector
--
-- r: true iff Ax = b
--
--------------------------------------------------------------------------------
linCheck(A, x, b) where {
```

```
  extern A :: IntMatrix
  extern x :: RatVector
  extern b :: IntVector

  #include "dclib.i"
  #include "saclib.i"

  type IntVector = [Integer]
  type IntMatrix = [IntVector]
  type RatVector = [Rational]

  -- returns true iff Ax = b
  linCheck :: (IntMatrix, RatVector, IntVector) -> Bool
  linCheck(A, x, b) = comp(cand, True) [ okay |
    Ai <- A, bi <- b,
    okay = RNCOMP(RNINT(bi), yi) == 0,
    yi = irvprod(Ai, x) ]

  -- integer rational vector product
  irvprod :: (IntVector, RatVector) -> Rational
  irvprod(a, b) =
    rcomp(RNSUM, RNZERO) [ RNPROD(RNINT(ai), bi) | ai <- a, bi <- b ]

  -- would be nice to have this constant in SACLIB
  RNZERO = RNINT(DTOI(0))

}
--------------------------------------------------------------------------------
-- end of linCheck.d
--------------------------------------------------------------------------------

/*
 * Generated by DC
 */

#define DC_SACLIB
#undef DC_NOINLINE
#undef DC_UNIX
#undef DC_REAL

#include "dc.h"

/*
 * type definitions
 */

typedef WORD LIST_311;
typedef WORD TUPLE_312;
... omitted ...
typedef WORD TUPLE_513;

/*
 * external functions and constants
 */

Rational RNQ(Rational, Rational);
Integer IEXP(Integer, INT);
... omitted ...
void IPOWER(Integer, Integer, Integer *, INT *);

/*
 * local function prototypes
 */
```

```
static inline Rational loop_548(IntegerList_91, RatVector_95, Rational);
static inline BOOL loop_554(IntMatrix_94, IntegerList_91, RatVector_95,
  Rational);

/*
 * local function prototypes
 */

static inline Rational loop_548(IntegerList_91 ail_543, RatVector_95 bil_544,
  Rational _547)
{
  while (!ISNIL(ail_543))
    if (ISNIL(bil_544))
      return (_547);
    else {
      _547 = RNSUM(RNPROD(RNINT(HEAD(ail_543)), HEAD(bil_544)), _547);
      bil_544 = TAIL(bil_544);
      ail_543 = TAIL(ail_543);
    }
  return (_547);
}

static inline BOOL loop_554(IntMatrix_94 Ail_549, IntegerList_91 bil_550,
  RatVector_95 x_556, Rational RNZERO_557)
{
  while (!ISNIL(Ail_549))
    if (ISNIL(bil_550))
      return (TRUE);
    else if (RNCOMP(RNINT(HEAD(bil_550)), loop_548(HEAD(Ail_549), x_556,
      RNZERO_557)) == 0) {
      bil_550 = TAIL(bil_550);
      Ail_549 = TAIL(Ail_549);
    } else
      return (FALSE);
  return (TRUE);
}

/*
 * external entry function
 */

BOOL linCheck(IntMatrix_94 A, RatVector_95 x, IntegerList_91 b)
{
  return (loop_554(A, b, x, RNINT(DTOI(0))));
}

/*
 * End of file
 */
```

C.1.4 linRand.d

```
--------------------------------------------------------------------------------
--
-- (A, b) = linRand(n, l, c)
--
-- Generate a random linear equation system
--
-- n: dimension of equation system
-- l: length of random integers in bits
--
-- (A, b): a system of dimension n with integer entries of at most l bits
--
```

```
--------------------------------------------------------------------------------
(A, b) where {

  extern n :: Int
  extern l :: Int

  #include "dclib.i"
  #include "saclib.i"

  -- that's a non-functional hack, since the compiler might "optimize" this
  -- code treating IRAND(l) as a constant (fortunately it does not yet :-)
  A = list [ Ai | i <- 1..n, Ai = list [ IRAND(l) | i <- 1..n ] ]
  b = list [ IRAND(l) | i <- 1..n ]

}
--------------------------------------------------------------------------------
-- end of linRand.d
--------------------------------------------------------------------------------

/*
 * Generated by DC
 */

#define DC_SACLIB
#undef DC_NOINLINE
#undef DC_UNIX
#undef DC_REAL

#include "dc.h"

/*
 * type definitions
 */

typedef WORD LIST_300;
typedef WORD TUPLE_301;
... omitted ...
typedef WORD TUPLE_512;

/*
 * external functions and constants
 */

INT MDCRA(INT, INT, INT, INT, INT);
INT DNOT(INT);
... omitted ...
Integer IBCOEF(INT, INT);

/*
 * local function prototypes
 */

static inline IntegerList_91 loop_520(INT, INT, INT);
static inline IntegerList_91 loop_514(INT, INT, INT);
static inline LIST_507 loop_517(INT, INT, INT);

/*
 * local function prototypes
 */

static inline void loop_547(INT i_291, INT n_521, INT l_522,
  IntegerList_91 * _548)
{
  while (i_291 <= n_521) {
    _548 = RCONS(IRAND(l_522), _548);
```

```
    i_291 = (i_291 + 1);
  }
  *_548 = NIL;
  return;
}

static inline IntegerList_91 loop_520(INT i_291, INT n_521, INT l_522)
{
  IntegerList_91 _548;
  loop_547(i_291, n_521, l_522, &_548);
  return (_548);
}

static inline void loop_549(INT i_290, INT n_527, INT l_528,
  IntegerList_91 * _550)
{
  while (i_290 <= n_527) {
    _550 = RCONS(IRAND(l_528), _550);
    i_290 = (i_290 + 1);
  }
  *_550 = NIL;
  return;
}

static inline IntegerList_91 loop_514(INT i_290, INT n_527, INT l_528)
{
  IntegerList_91 _550;
  loop_549(i_290, n_527, l_528, &_550);
  return (_550);
}

static inline void loop_551(INT i_288, INT n_524, INT l_525, LIST_507 * _552)
{
  while (i_288 <= n_524) {
    _552 = RCONS(loop_514(1, n_524, l_525), _552);
    i_288 = (i_288 + 1);
  }
  *_552 = NIL;
  return;
}

static inline LIST_507 loop_517(INT i_288, INT n_524, INT l_525)
{
  LIST_507 _552;
  loop_551(i_288, n_524, l_525, &_552);
  return (_552);
}

/*
 * external entry function
 */

void linRand(INT n, INT l, LIST_507 * _554, IntegerList_91 * _555)
{
  {
    *_554 = loop_517(1, n, l);
    *_555 = loop_520(1, n, l);
    return;
  }
}

/*
 * End of file
 */
```

C.1.5 linDets.c

```
/******************************************************************************
 *
 * (d, y) = linDets(A, b, n, m)
 *
 * Compute modular determinants
 *
 * A: an Integer matrix of dimension n that is regular
 * b: an Integer vector of length n
 * n: a Digit
 * m: a Digit
 *
 * d: det(A) mod Zm
 * y: list of yi = det(Ai) mod Zm where Ai = A with column i replaced by b
 *
 *****************************************************************************/

#include <saclib.h>

/******************************************************************************
 *
 * Return Am = A mod m and bm = b mod m where Am and bm are arrays!
 *
 *****************************************************************************/
static void linMap(Word A, Word b, Word n, Word m, Word *Am, Word *bm) {

  Word i, j, A_, b_;

  A_ = GCAMALLOC(n, GC_CHECK);
  for (i = 0; i < n; i++)
    GCASET(A_, i, GCAMALLOC(n, GC_CHECK));
  b_ = GCAMALLOC(n, GC_CHECK);

  for (i = 0; i < n; i++) {
    Word Ai = FIRST(A);
    Word A_i = GCAGET(A_, i);
    for (j = 0; j < n; j++) {
      GCASET(A_i, j, MDHOM(m, FIRST(Ai)));
      Ai = RED(Ai);
    }
    GCASET(b_, i, MDHOM(m, FIRST(b)));
    A = RED(A);
    b = RED(b);
  }

  *Am = A_;
  *bm = b_;

}

/******************************************************************************
 *
 * Compute d = det(A) and yi = det(Ai) in Zm by LU decomposition.
 *
 *****************************************************************************/
void linDets(Word A, Word b, Word n, Word m, Word *d, Word *y) {

  Word Am, bm, ym;
  Word Amk, Amkk, Ami, bmi;
  Word i, j, k, p;
  Word det, tmp;

  linMap(A, b, n, m, &Am, &bm);
```

```
  det = 1;
  for (k = 0; k < n-1; k ++) {

    p = k;
    while(p < n && GCAGET(GCAGET(Am, p), k) == 0) p++;
    if (p == n) {
      *d = 0;
      return;
    }

    if (p != k) {
      tmp = GCAGET(Am, k); GCASET(Am, k, GCAGET(Am, p)); GCASET(Am, p, tmp);
      tmp = GCAGET(bm, k); GCASET(bm, k, GCAGET(bm, p)); GCASET(bm, p, tmp);
      det = MDNEG(m, det);
    }

    Amk  = GCAGET(Am, k);
    Amkk = GCAGET(Amk, k);
    det = MDPROD(m, det, Amkk);

    for (i = k+1; i < n; i++) {
      Ami = GCAGET(Am, i);
      GCASET(Ami, k, MDQ(m, GCAGET(Ami, k), Amkk));
      for (j = k+1; j < n; j++) {
        GCASET(Ami, j, MDDIF(m, GCAGET(Ami, j),
                             MDPROD(m, GCAGET(Ami, k), GCAGET(Amk, j))));
      }
    }

  }

  det = MDPROD(m, det, GCAGET(GCAGET(Am, n-1), n-1));
  if (det == 0) {
     *d = 0;
     return;
   }

  for (i = 0; i < n; i++) {
    Ami = GCAGET(Am, i);
    bmi = GCAGET(bm, i);
    for (j = 0; j < i; j++) {
      bmi = MDDIF(m, bmi, MDPROD(m, GCAGET(Ami, j), GCAGET(bm, j)));
    }
    GCASET(bm, i, bmi);
  }

  for (i = n-1; i >= 0; i--) {
    Ami = GCAGET(Am, i);
    tmp = GCAGET(bm, i);
    for (k = i+1; k  < n; k++)
      tmp = MDDIF(m, tmp, MDPROD(m, GCAGET(Ami, k), GCAGET(bm, k)));
    GCASET(bm, i, MDQ(m, tmp, GCAGET(Ami, i)));
  }

  ym = NIL;
  for (i = n-1; i >= 0; i--)
    ym = COMP(MDPROD(m, det, GCAGET(bm, i)), ym);

  *d = det;
  *y = ym;

}
```

C.1.6 linMain.c

```
/******************************************************************************
 *
 * linMain n l c p
 *
 * solve a random linear equation system
 *
 * n: number of dimensions
 * l: length of random integers in bit
 * c: 1 iff result is to be checked
 * p: 1 iff system is to be printed
 *
 *****************************************************************************/

#include <saclib.h>
#include <stdlib.h>

extern void linRand(Word n, Word l, Word *A, Word *b);
extern Word linSolve(Word A, Word b, Word n);
extern Word linCheck(Word A, Word x, Word b);

/******************************************************************************
 *
 * Get command line arguments
 *
 *****************************************************************************/
void linArgs(int argc, char **argv, Word *n, Word *l, Word *c, Word *p){

  if (argc < 3) {
    SWRITE("syntax: linsolve n l [ c [ p ] ]\n");
    exit(-1);
  }
  *n = atoi(argv[1]);
  *l = atoi(argv[2]);
  if (argc < 4)
    *c = 0;
  else
    *c = atoi(argv[3]);
  if (argc < 5)
    *p = 0;
  else
    *p = atoi(argv[4]);

}

/*****************************************************************************
 *
 * print integer list
 *
 ****************************************************************************/
void linPrintI(Word x) {

  SWRITE("\n[");
  while (!ISNIL(x)) {
    IWRITE(FIRST(x));
    x = RED(x);
    if (!ISNIL(x)) SWRITE(",");
  }
  SWRITE("]");

}

/*****************************************************************************
 *
```

```
 * print rational list
 *
 ***************************************************************************/
void linPrintR(Word x) {

  SWRITE("\n[");
  while (!ISNIL(x)) {
    RNWRITE(FIRST(x));
    x = RED(x);
    if (!ISNIL(x)) SWRITE(",");
  }
  SWRITE("]");

}

/****************************************************************************
 *
 * Print result
 *
 ***************************************************************************/
void linPrint(Word A, Word x, Word b) {

  while (!ISNIL(A)) {
    linPrintI(FIRST(A));
    A = RED(A);
  }
  SWRITE("\n");
  linPrintI(b);
  SWRITE("\n");
  linPrintR(x);
  SWRITE("\n");

}

/*****************************************************************************
 *
 * Main program
 *
 ****************************************************************************/
Word sacMain(int argc, char **argv) {

  Word n, l, c, p;
  Word A, x, b;
  Word time;

  linArgs(argc, argv, &n, &l, &c, &p);
  linRand(n, l, &A, &b);

  time = CLOCK();
  x = linSolve(A, b, n);
  time = CLOCK()-time;

  IWRITE(time);
  SWRITE(" ms\n");

  if (p) linPrint(A, x, b);
  if (c && !linCheck(A, x, b)) SWRITE("Something went wrong.\n");

  return(0);

}
```

C.2 Parallel Linear Equation Solver

This section presents the parallel version of `linSolve.d` and the corresponding target code.

```
----------------------------------------------------------------------------
--
-- x = linSolve(A, b, n)
--
-- Solve a linear equation system ("parallel" version)
--
-- A: an Integer matrix of dimension n that is regular
-- b: an Integer vector of length n
-- n: a Digit
--
-- x: the unique Rational-Vector of length n such that Ax = b
--
----------------------------------------------------------------------------
linSolve(A, b, n, PRIME) where {

  -- program arguments (PRIME declared in saclib.i)
  extern A :: IntMatrix
  extern b :: IntVector
  extern n :: Digit

  -- libraries
  #include "dclib.i"
  #include "saclib.i"

  -- type names
  type IntVector = [Integer]
  type IntMatrix = [IntVector]
  type RatVector = [Rational]

  -- solve Ax=b for x
  linSolve :: (IntMatrix, IntVector, Digit, DigitList) -> RatVector
  linSolve(A, b, n, PLIST) = x where

    B = bound(A, b, n);

    base = ([], [], [], [], [])
    (dkyk, pk, Pk, ik, PREST) = comp(compose, base) [ (dy, p, P, PREST) |
        p:PREST <- PLIST, P = DTOI(1)
      ; ICOMP(P, B) == -1
      ; P' = IPROD(DTOI(p), P),
        dy = linDets(A, b, n, p)@ ]                      -- ANNOTATION

    d = crad(pk, ik, Pk, dkyk)@                          -- ANNOTATION
    ys = list [ cray(pk, ik, Pk, dkyk, i)@ | i <- 1..n ]@  -- ANNOTATION
    x0 = list [ RNRED(y, d)@ | y <- ys ]@                -- ANNOTATION

    x = if comp(cand, True) [ d /= 0 | dy <- dkyk, (d,y) = dy ]
          then x0
          else linSolve(A, b, n, PREST)

  -- prime bound: P = 2*n!*m^n where m = max(Aij, bi)
  bound :: (IntMatrix, IntVector, Digit) -> Integer
  bound(A, b, n) = P where
    m0 = rcomp(IMAX, DTOI(0)) [ IABSF(x) | x <- b ]
    m1 = rcomp(IMAX, m0) [ IABSF(x) | l <- A | x <- l ]
    P = IPROD(DTOI(2), IPROD(IFACTL(n), IEXP(m1, n)))

  -- modular computation of determinants
  extern linDets :: (IntMatrix, IntVector, Digit, Digit) -> (Digit, DigitList)
```

```
  -- auxiliary types
  type DigitPair = (Digit, DigitList)
  type DIL = ([DigitPair], DigitList, IntegerList, DigitList, DigitList)

  -- composition of modular determinants, primes, products and inverses
  compose :: ((DigitPair, Digit, Integer, DigitList), DIL) -> DIL
  compose((dy, p, P, PREST), (dkyk, pk, Pk, ik, PREST0)) =
    (dy:dkyk, p:pk, P:Pk, i:ik, PREST1) where
      i = MDINV(p, MDHOM(p, P))
      PREST1 = case PREST0 of {[] => PREST; h:t => PREST0}

  -- Chinese remainder algorithm on d component
  crad :: (DigitList, DigitList, IntegerList, [DigitPair]) -> Integer
  crad(ms, mis, Ms, as) = rcomp(cra2, DTOI(1)) [ (m, mi, M, d) |
      m <- ms, mi <- mis, M <- Ms, a <- as, (d,yi) = a
    ; d /= 0 ]

  -- Chinese remainder algorithm on yi component
  cray :: (DigitList, DigitList, IntegerList, [DigitPair], Int) -> Integer
  cray(ms, mis, Ms, as, i) = rcomp(cra2, DTOI(1)) [ (m, mi, M, y) |
      m <- ms, mi <- mis, M <- Ms, a <- as, (d,yi) = a
    ; d /= 0
    ; y = nth(i, yi) ]

  -- Chinese remainder algorithm on two arguments
  cra2 :: ((Digit, Digit, Integer, Digit), Integer) -> Integer
  cra2((m, mi, M, a), A) = MIDCRA(M, m, mi, A, a)

}

/*
 * Generated by D2C
 */

#undef DC_SACLIB
#undef DC_NOINLINE
#undef DC_UNIX
#undef DC_REAL

#include "dc.h"

/*
 * type definitions
 */

typedef WORD LIST_408;          /* [*a_25] */
... omitted ...
typedef WORD TASK_1558;         /* Task(([Int], [Int], [Integer],
  [Task((Int, [Int]))], Integer, Int)) */

/*
 * local function prototypes
 */

static __inline__ Rational fun_1264(LIST_917, TASK_916);
static __inline__ Integer loop_993(IntList_44, IntList_44, IntegerList_108,
  LIST_854, Integer);
static __inline__ BOOL loop_965(LIST_854);
static __inline__ Integer fun_1004(LIST_854, IntList_44, IntegerList_108,
  IntList_44);
static __inline__ RatVector_124 listsync_840(LIST_816);
static __inline__ Integer loop_679(IntegerList_108, Integer);
static __inline__ Integer loop_689(IntegerList_108, Integer);
static __inline__ Integer loop_685(IntMatrix_123, Integer);
static __inline__ TUPLE_870 loop_872(IntList_44, Integer, IntMatrix_123,
```

```
  IntegerList_108, INT, Integer);
static __inline__ INT nth_646(INT, IntList_44);
static __inline__ Integer loop_1376(IntList_44, IntList_44, IntegerList_108,
  LIST_854, Integer, INT);
static __inline__ Integer fun_1149(INT, LIST_854, IntList_44,
  IntegerList_108, IntList_44);
static __inline__ LIST_816 fun_1513(LIST_917, TASK_916);
static __inline__ LIST_816 fun_1029(INT, LIST_854, IntList_44,
  IntegerList_108, IntList_44);
static __inline__ LIST_917 loop_973(INT, INT, LIST_854, IntList_44,
  IntegerList_108, IntList_44);
static __inline__ LIST_917 fun_985(INT, LIST_854, IntList_44, IntegerList_108,
  IntList_44);
static __inline__ TASK_1036 fun_1293(IntMatrix_123, IntegerList_108, INT,
  TUPLE_870);

/*
 * local function definitions
 */

static __inline__ Rational fun_1264(LIST_917 _1265, TASK_916 _1266)
{
  TASK_916 _1599 = HEAD(_1265);
  Integer _1709 = WAIT(_1599);
  Integer _1710 = WAIT(_1266);
  return (RNRED(_1709, _1710));
}

static __inline__ Integer loop_993(IntList_44 _994, IntList_44 _995,
  IntegerList_108 _996, LIST_854 _997, Integer _998)
{
  while (!ISNIL(_994))
    if (ISNIL(_995))
      return (_998);
    else if (ISNIL(_996))
      return (_998);
    else if (ISNIL(_997))
      return (_998);
    else {
      DigitPair_127 _1711 = WAIT(HEAD(_997));
      INT d_1220 = ELEM1(_1711);
      if (d_1220 != 0) {
        _998 = MIDCRA(HEAD(_996), HEAD(_994), HEAD(_995), _998, d_1220);
        _997 = TAIL(_997);
        _996 = TAIL(_996);
        _995 = TAIL(_995);
        _994 = TAIL(_994);
      } else
        return (_998);
    }
  return (_998);
}

static __inline__ BOOL loop_965(LIST_854 _966)
{
  while (!ISNIL(_966)) {
    DigitPair_127 _1712 = WAIT(HEAD(_966));
    if (ELEM1(_1712) != 0)
      _966 = TAIL(_966);
    else
      return (FALSE);
  }
  return (TRUE);
}
```

```
static __inline__ Integer fun_1004(LIST_854 dkyk_1005, IntList_44 pk_1006,
  IntegerList_108 Pk_1007, IntList_44 ik_1008)
{
  return (loop_993(pk_1006, ik_1008, Pk_1007, dkyk_1005, DTOI(1)));
}

static __inline__ void listsync_1727(LIST_816 _841, RatVector_124 * _1728)
{
  while (!ISNIL(_841)) {
    Rational _1713 = WAIT(HEAD(_841));
    _1728 = RCONS(_1713, _1728);
    _841 = TAIL(_841);
  }
  *_1728 = NIL;
  return;
}

static __inline__ RatVector_124 listsync_840(LIST_816 _841)
{
  RatVector_124 _1728;
  listsync_1727(_841, &_1728);
  return (_1728);
}

static __inline__ Integer loop_679(IntegerList_108 xl_676, Integer _678)
{
  while (!ISNIL(xl_676)) {
    _678 = IMAX(IABSF(HEAD(xl_676)), _678);
    xl_676 = TAIL(xl_676);
  }
  return (_678);
}

static __inline__ Integer loop_689(IntegerList_108 xl_686, Integer _688)
{
  while (!ISNIL(xl_686)) {
    _688 = IMAX(IABSF(HEAD(xl_686)), _688);
    xl_686 = TAIL(xl_686);
  }
  return (_688);
}

static __inline__ Integer loop_685(IntMatrix_123 ll_682, Integer _684)
{
  while (!ISNIL(ll_682)) {
    _684 = loop_689(HEAD(ll_682), _684);
    ll_682 = TAIL(ll_682);
  }
  return (_684);
}

static __inline__ void loop_1733(IntList_44 pl_873, Integer P_874,
  IntMatrix_123 A_875, IntegerList_108 b_876, INT n_877, Integer B_878,
  LIST_854 * _1734, IntList_44 * _1735, IntegerList_108 * _1736,
  IntList_44 * _1737, IntList_44 * _1738)
{
  if (ISNIL(pl_873)) {
    *_1734 = NIL;
    *_1735 = NIL;
    *_1736 = NIL;
    *_1737 = NIL;
    *_1738 = NIL;
    return;
  } else {
    INT p_879 = HEAD(pl_873);
```

```
    IntList_44 PREST_880 = TAIL(pl_873);
    if (ICOMP(P_874, B_878) == -1) {
      IntList_44 _1732;
      IntList_44 _1743;
      TASK_853 _1714 = START4(linDets, A_875, b_876, n_877, p_879);
      loop_1733(PREST_880, IPROD(DTOI(p_879), P_874), A_875, b_876, n_877,
        B_878, RCONS(_1714, _1734), RCONS(p_879, _1735), RCONS(P_874, _1736),
        RCONS(MDINV(p_879, MDHOM(p_879, P_874)), _1737), &_1743);
      if (ISNIL(_1743))
        _1732 = PREST_880;
      else
        _1732 = _1743;
      *_1738 = _1732;
      return;
    } else {
      *_1734 = NIL;
      *_1735 = NIL;
      *_1736 = NIL;
      *_1737 = NIL;
      *_1738 = NIL;
      return;
    }
  }
}

static __inline__ TUPLE_870 loop_872(IntList_44 pl_873, Integer P_874,
  IntMatrix_123 A_875, IntegerList_108 b_876, INT n_877, Integer B_878)
{
  IntList_44 _1734;
  IntList_44 _1735;
  IntegerList_108 _1736;
  IntList_44 _1737;
  LIST_854 _1738;
  loop_1733(pl_873, P_874, A_875, b_876, n_877, B_878, &_1734, &_1735,
    &_1736, &_1737, &_1738);
  return (TUPLE5(_1734, _1735, _1736, _1737, _1738));
}

static __inline__ INT nth_646(INT n_647, IntList_44 l_648)
{
  while (!ISNIL(l_648))
    if (n_647 == 1)
      return (HEAD(l_648));
    else {
      l_648 = TAIL(l_648);
      n_647 = (n_647 - 1);
    }
  ERROR("nth");
}

static __inline__ Integer loop_1376(IntList_44 _1377, IntList_44 _1378,
  IntegerList_108 _1379, LIST_854 _1380, Integer _1381, INT _1382)
{
  while (!ISNIL(_1377))
    if (ISNIL(_1378))
      return (_1381);
    else if (ISNIL(_1379))
      return (_1381);
    else if (ISNIL(_1380))
      return (_1381);
    else {
      TASK_853 a_1569 = HEAD(_1380);
      DigitPair_127 _1715 = WAIT(a_1569);
      if (ELEM1(_1715) != 0) {
        _1381 = MIDCRA(HEAD(_1379), HEAD(_1377), HEAD(_1378), _1381,
```

```
        nth_646(_1382, ELEM2(_1715)));
        _1380 = TAIL(_1380);
        _1379 = TAIL(_1379);
        _1378 = TAIL(_1378);
        _1377 = TAIL(_1377);
      } else
        return (_1381);
    }
  return (_1381);
}

static __inline__ Integer fun_1149(INT _1150, LIST_854 _1151,
  IntList_44 _1152, IntegerList_108 _1153, IntList_44 _1154)
{
  return (loop_1376(_1152, _1154, _1153, _1151, DTOI(1), _1150));
}

static __inline__ void fun_1747(LIST_917 _1514, TASK_916 _1515,
  LIST_816 * _1748)
{
  while (TRUE) {
    LIST_917 _1617 = TAIL(_1514);
    if (ISNIL(_1617)) {
      *_1748 = NIL;
      return;
    } else {
      TASK_815 _1716 = START2(fun_1264, _1617, _1515);
      _1748 = RCONS(_1716, _1748);
      _1514 = _1617;
    }
  }
}

static __inline__ LIST_816 fun_1513(LIST_917 _1514, TASK_916 _1515)
{
  LIST_816 _1748;
  fun_1747(_1514, _1515, &_1748);
  return (_1748);
}

static __inline__ LIST_816 fun_1029(INT n_1030, LIST_854 dkyk_1031,
  IntList_44 pk_1032, IntegerList_108 Pk_1033, IntList_44 ik_1034)
{
  TASK_992 _1719 = START5(fun_985, n_1030, dkyk_1031, pk_1032, Pk_1033,
    ik_1034);
  TASK_916 _1718 = START4(fun_1004, dkyk_1031, pk_1032, Pk_1033, ik_1034);
  LIST_917 _1720 = WAIT(_1719);
  if (ISNIL(_1720))
    return (NIL);
  else {
    TASK_815 _1717 = START2(fun_1264, _1720, _1718);
    return (CONS(_1717, fun_1513(_1720, _1718)));
  }
}

static __inline__ void loop_1750(INT _974, INT _975, LIST_854 _976,
  IntList_44 _977, IntegerList_108 _978, IntList_44 _979, LIST_917 * _1751)
{
  while (_974 <= _975) {
    TASK_916 _1721 = START5(fun_1149, _974, _976, _977, _978, _979);
    _1751 = RCONS(_1721, _1751);
    _974 = (_974 + 1);
  }
  *_1751 = NIL;
  return;
```

```
}

static __inline__ LIST_917 loop_973(INT _974, INT _975, LIST_854 _976,
  IntList_44 _977, IntegerList_108 _978, IntList_44 _979)
{
  LIST_917 _1751;
  loop_1750(_974, _975, _976, _977, _978, _979, &_1751);
  return (_1751);
}

static __inline__ LIST_917 fun_985(INT n_986, LIST_854 dkyk_987,
  IntList_44 pk_988, IntegerList_108 Pk_989, IntList_44 ik_990)
{
  return (loop_973(1, n_986, dkyk_987, pk_988, Pk_989, ik_990));
}

static __inline__ TASK_1036 fun_1293(IntMatrix_123 A_1294,
  IntegerList_108 b_1295, INT n_1296, TUPLE_870 _1297)
{
  while (TRUE) {
    LIST_854 _1754;
    IntList_44 _1755;
    IntegerList_108 _1756;
    IntList_44 _1757;
    IntList_44 _1758;
    loop_1733(ELEM5(_1297), DTOI(1), A_1294, b_1295, n_1296, IPROD(DTOI(2),
      IPROD(IFACTL(n_1296), IEXP(loop_685(A_1294, loop_679(b_1295, DTOI(0))),
      n_1296))), &_1754, &_1755, &_1756, &_1757, &_1758);
    if (loop_965(_1754)) {
      TASK_1036 _1722 = START5(fun_1029, n_1296, _1754, _1755, _1756, _1757);
      return (_1722);
    } else
      _1297 = TUPLE5(_1754, _1755, _1756, _1757, _1758);
  }
}

/*
 * external entry function
 */

RatVector_124 dcmain(IntMatrix_123 A, IntegerList_108 b, INT n)
{
  LIST_816 _1707;
  TASK_1036 _1759;
  LIST_854 _1760;
  IntList_44 _1761;
  IntegerList_108 _1762;
  IntList_44 _1763;
  IntList_44 _1764;
  loop_1733(PRIME, DTOI(1), A, b, n, IPROD(DTOI(2), IPROD(IFACTL(n),
    IEXP(loop_685(A, loop_679(b, DTOI(0))), n))), &_1760, &_1761, &_1762,
      &_1763, &_1764);
  if (loop_965(_1760)) {
    TASK_1036 _1708 = START5(fun_1029, n, _1760, _1761, _1762, _1763);
    _1759 = _1708;
  } else
    _1759 = fun_1293(A, b, n, TUPLE5(_1760, _1761, _1762, _1763, _1764));
  _1707 = WAIT(_1759);
  return (listsync_840(_1707));
}

/*
 * End of file
 */
```