# *Compiling a Para-Functional Language to Parallel PACLIB C*[*]

*Wolfgang Schreiner*
*Research Institute for Symbolic Computation (RISC-Linz)*
*Johannes Kepler University, Linz, Austria*
`Wolfgang.Schreiner@risc.uni-linz.ac.at`

August 8, 1994

**Abstract**

*We describe the design and implementation of* pD*, a parallel variant of the small functional language D. In particular, we show 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 program written by a human programmer. The generated code can be linked with the PACLIB kernel, the multi-processor runtime system for the computer algebra library SACLIB. This allows to develop parallel computer algebra applications in an elegant and efficient manner.*

[*]Supported by the FWF grant S5302-PHY "Parallel Symbolic Computation".

# Contents

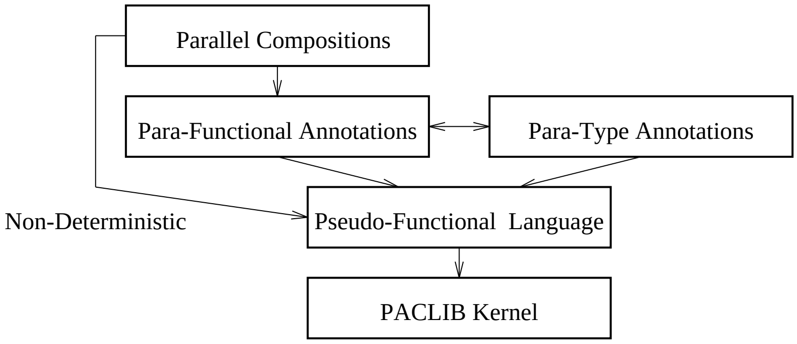

Figure 1: Para-Functional Interface

# 1 Introduction

In this report, we describe the design and implementation of pD, a parallel variant of the small functional language D 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 program written by a human programmer.

The generated code can be linked with the PACLIB kernel, the multi-processor runtime system for the computer algebra library SACLIB. 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 *efficiency* and *flexibility*:

1. pD is intended for parallel programming. The only motivation for developing a parallel program 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 variants.
2. pD is intended for computer algebra. This area is based on dynamic data structures and algorithms that demand very flexible forms of parallelism. Pure data parallelism (SIMD/SPMD) is not sufficient; only those approaches are appropriate that may cope with tasks with extremely varying execution times.

We chose the functional approach for high-level parallel programming since computer algebra is based on the concept of mathematical functions and most computer algebra programs are therefore 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.

Figure 1 demonstrates the structure of the pD system.

1. **Pseudo-Functional Core Language** (Section 2) On the lowest level, pD supports the additional types `Task` and `Stream` and functions for the creation and synchronization of tasks. This level also supports non-functional features (such as non-determinism and task abortion).

2. **Non-Deterministic Compositions** (Section 3) The non-deterministic features are embedded into a referential transparent framework using a high-level form of compositions.

3. **Para-Functional Language** (Section 4) Using a para-functional annotation language (on expressions and types), the user directs the compiler to generate task creations and synchronizations.

4. **Parallel Compositions** (Section 5) The programmer may use parallel variants of high-level compositional forms as a sort of "templates" for parallel programming; the compiler translates these compositions into programs in the para-functional language.

The application of the para-functional system to the parallelization of several example programs is finally demonstrated in Section 6.

As summarized in [Sch93b], there are many related research projects on the application of functional languages for parallel programming; yet not many of them have reached practical significance:

SISAL [FCO90] is the most successful example of a functional language used for parallel programming. Compilers for this language produce high quality target code for a variety of parallel architectures. The language is mainly intended for scientific (i.e. numerical) applications traditionally implemented in Fortran. While the language definition itself is rather general, many implementations put restrictions on data structures and/or exploited parallelism.

Other parallel functional languages mainly for numerical applications are Lucid [WA85] and Crystal [Che85]. Several data-parallel versions of functional languages have been implemented in particular for (but not restricted to) SIMD architectures [SJH86, Mer91, Hil93, Ble93].

In parallel Lisp dialects based on futures [Hal85, ZHHL87, GG89], the programmer may explicitly start parallel tasks. These tasks are transparently synchronized which requires runtime checks at all (strict) primitive operations (which may be embedded into a general runtime type checking mechanism).

The term para-functional programming was coined by [Hud86] who introduced parallelization annotations in the lazy functional language parAlfl. The annotation language was refined in [Hud91]. Similar annotations were used in [KG89, Kel89, NSEP91].

Skeleton-based approaches [Col89, DFH+93, Rab93, DOPV94] use higher order functions to express patterns of computations for which the compiler is able to generate parallel code. Most systems are based on fixed sets of templates that cannot be freely extended by the user. In general the computations in different program phases (expressed by compositions of templates) may not overlap but require synchronization between each stage.

The pH project [NAH93] (from which we have borrowed the name pD) aims at a parallel implementation of the lazy functional language HASKELL using features from Id [Nik91]. In particular, an implementation for a workstation cluster is under way. Any parallel Haskell variant has to cope with the considerable sequential runtime overhead of lazy evaluation.

The *original features* of our work 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 elaborated in Section 4. 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

| | | | |
|---|---|---|---|
| *exp* | → | . . . see [Sch93a] . . . | 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 Creation |
| | \| | `fstart` ( *f* , *id* , $exp_1$ , . . . , $exp_n$ ) | Forward Start |
| *patterns* | → | `[ ] =>` $exp_n$ ; $id_h$ : $id_t$ `=>` $exp_c$ | Stream Patterns |

Figure 2: Abstract Syntax

is statically typed (i.e. no runtime task checks are required). Still by agressive deferring of synchronizations most of the parallelism in functional languages based on futures or non-strict data-flow is preserved.

The first issue distinguishes our approach 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 (but then very efficient) 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 for generality by inserting at strict basis operations synchronization checks or rely on hardware support.

We have attempted to provide a system that generates parallel C code whose efficiency and whose structure matches that of parallel code written by a human programmer. Concerning efficiency, in most cases there should be no need to rewrite programs on the C level. Concerning structure, the human should be able to read and understand the target code for analyzing and optimizing the program behavior (performance debugging).

# 2 The Pseudo-Functional Core Language

In this section we define the core language that serves as the functional interface to the parallel runtime system (i.e. the PACLIB kernel). Those features that exceed the purely functional framework (forward tasks, task abortion and non-determinism) are described at the end of this section.

## 2.1 Abstract Syntax

The parallel core language extends the basic core language defined in [Sch93a] 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 when its result is actually required. In other words, the execution of tasks is *non-strict*.
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 2 gives the abstract syntax of these extensions. Except for `any`, `stop`, `forward`, `fstart` (described at the end of this section), the extended semantics fits nicely into the framework described in [Sch93a]. We therefore only give the necessary extensions.

## 2.2 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 Section 5.

## 2.3 Semantic Algebras

In the following subsections, 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.

The domain 'Value' is extended by

| | | |
|---|---|---|
| Task = | *Unit* $\rightarrow$ Value | Task |
| SNil = | *Unit* | Empty Stream |
| Stream = | Value $\times$ Value | Non-Empty Stream |
| Value = | $(\ldots$see [Sch93a]$\ldots$ + Task + SNil + Stream$)_\perp$ | Denotable Value |

and correspondingly the domain 'Type':

| | | |
|---|---|---|
| TaskType = | Type | Task |
| SNilType = | *Unit* | Empty Stream |
| StreamType = | Type | Non-Empty Stream |
| Type = | (...see [Sch93a] ... + TaskType + SNilType + StreamType)$_\perp$ | Denotable Type |

The environment domains 'ValueEnv' and 'TypeEnv' are constructed as usual:

| | | |
|---|---|---|
| ValueEnv = | (Identifier $\rightarrow$ Value)$_\perp$ | Value Environment |
| TypeEnv = | (Identifier $\rightarrow$ Type)$_\perp$ | Type Environment |

The construction mk$T$(...) that embedds a type into some union type is always *strict* in its arguments e.g. mkStream($\perp, t$) = $\perp$[1].

## 2.4 Typing

The type compatibility function 'eqType' must be extended as follows:

eqType : Type $\times$ Type $\rightarrow$ Bool
eqType($T_1, T_2$) =
  $T_1$ = mkErrorType() $\vee$ $T_2$ = mkErrorType() $\rightarrow$ true;
  case $T_1$ of
    ...
    isTaskType($t_1$) $\rightarrow$
      case $T_2$ of
        isTaskType($t_2$) $\rightarrow$ eqType($t_1, t_2$)
        otherwise $\rightarrow$ false
    isSNilType() $\rightarrow$
      case $T_2$ of
        isSNilType() $\rightarrow$ true
        isStreamType($t$) $\rightarrow$ true
        otherwise $\rightarrow$ false
    isStreamType($t_1$) $\rightarrow$
      case $T_2$ of
        isSNilType() $\rightarrow$ true
        isStreamType($t_2$) $\rightarrow$ eqType($t_1, t_2$)
        otherwise $\rightarrow$ false

Likewise 'joinType' is extended:

joinType : Type $\times$ Type $\rightarrow$ Type
joinType($T_1, T_2$) =
  $T_1$ = mkErrorType() $\rightarrow$ $T_2$;
  $T_2$ = mkErrorType() $\rightarrow$ $T_1$;
  case $T_1$ of
    ...
    isTaskType($t_1$) $\rightarrow$
      case $T_2$ of isTaskType($t_2$) $\rightarrow$ mkTaskType(joinType($t_1, t_2$))
    isSNilType() $\rightarrow$
      case $T_2$ of
        isSNilType() $\rightarrow$ mkSNilType()
        isStreamType($t$) $\rightarrow$ $T_2$
    isStreamType($t_1$) $\rightarrow$

[1] The same restriction also applied to the definitions in [Sch93a] but was not explicitly mentioned there.

$$
\begin{array}{l}
\text{case } T_2 \text{ of} \\
\quad \text{isSNilType}() \rightarrow T_1 \\
\quad \text{isStreamType}(t_2) \rightarrow \text{mkType}(\text{joinType}(t_1, t_2))
\end{array}
$$

### 2.5 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.

### 2.6 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`(mkTask($\lambda x.\perp$)) $= \perp$

### 2.7 Empty Stream

$$
\begin{array}{l}
\mathbf{T}[\![\texttt{[ ]}_\texttt{S}]\!] = \underline{\lambda} e.\text{mkSNilType}() \\
\mathbf{E}[\![\texttt{[ ]}_\texttt{S}]\!] = \underline{\lambda} e.\text{mkSNil}()
\end{array}
$$

### 2.8 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.

### 2.9 Stream Selection

$$
\begin{array}{l}
\mathbf{T}[\![\texttt{scase}\ exp_s\ \texttt{of [ ] =>}\ exp_n\ \texttt{;}\ id_h\texttt{:}id_t\ \texttt{=>}\ exp_c]\!] = \underline{\lambda} e. \\
\quad \text{case } \mathbf{T}[\![exp_s]\!]e \text{ of} \\
\quad\quad \text{isErrorType}() \rightarrow \text{joinType}(\mathbf{T}[\![exp_n]\!]e, \mathbf{T}[\![exp_c]\!]e); \\
\quad\quad \text{isSNilType}() \rightarrow \text{joinType}(\mathbf{T}[\![exp_n]\!]e, \mathbf{T}[\![exp_c]\!]e);
\end{array}
$$

$$
\begin{array}{l}
\text{isStreamType}(T) \ \rightarrow \\
\quad \text{joinType}(\mathbf{T}[\![exp_n]\!]e, \\
\quad\quad \mathbf{T}[\![exp_c]\!][[\![id_h]\!] \leftarrow T, [\![id_t]\!] \leftarrow \ \text{mkTaskType}(\text{mkStreamType}(T))] \ e
\end{array}
$$

As for the `case` construct [Sch93a], `scase` cannot give types for $id_h$ and $id_t$ if $exp_s$ is the `error` constant. The variables must not be used then.

$$
\begin{array}{l}
\mathbf{E}[\![\texttt{scase}\ exp_s\ \texttt{of [ ] =>}\ exp_n\ \texttt{;}\ id_h : id_t\ \texttt{=>}\ exp_c]\!] = \underline{\lambda} e. \\
\quad \text{case } (\mathbf{E}[\![exp_s]\!]e) \text{ of} \\
\quad\quad \text{isSNil}() \rightarrow \mathbf{E}[\![exp_n]\!]e; \\
\quad\quad \text{isStream}(h, t) \rightarrow \ \mathbf{E}[\![exp_c]\!][[\![id_h]\!] \leftarrow h, [\![id_t]\!] \leftarrow t] \ e
\end{array}
$$

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.

## 2.10 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 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* [Wad91] where every identifier has at most one applied occurence. 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

$$\text{CValue} = \mathbb{N} \times \text{Value}$$
$$\text{CValueEnv} = \text{Ident} \rightarrow \text{CValue}$$

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

$$\mathbf{TA}\left[\!\!\left[\begin{array}{l} \ldots \\ x\ \texttt{=}\ \ldots \\ \ldots \\ \ldots\ x\ \ldots\ x\ \ldots \\ \ldots \end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l} \ldots \\ x\ \texttt{=}\ \ldots \\ (x_1\texttt{,}\ x_2\texttt{)}\ \texttt{=}\ \texttt{copy(}x\texttt{)} \\ \ldots \\ \ldots\ x_1\ \ldots\ x_2\ \ldots \\ \ldots \end{array}\right]\!\!\right]$$

The definition of constants (see [Sch93a]) has to be extended by

$$\begin{array}{l}
\mathbf{DA}[\![ id_1^{T_1}\ \texttt{=}\ exp_1\ \texttt{;}\ \ldots\texttt{;}\ \texttt{(}id_a\texttt{,}\ id_b\texttt{)}\ \texttt{=}\ \texttt{copy(}exp\texttt{);}\ \ldots\ \texttt{;}\ id_n^{T_n}\ \texttt{=}\ exp_n ]\!] = \underline{\lambda} e_0. \\
\quad \text{let } e_1 = [[\![ id_1 ]\!] \leftarrow \mathbf{EA}[\![ exp_1 ]\!] e_0]\ e_0 \text{ in} \\
\quad \ldots \\
\quad \text{let } e_{i+1} = [[\![ id_a ]\!] \leftarrow \mathbf{EA}[\![ \texttt{copy(}exp\texttt{)} ]\!] e_i]\ e_i \text{ in} \\
\quad \text{let } e_{i+2} = [[\![ id_b ]\!] \leftarrow \mathbf{EA}[\![ \texttt{copy(}exp\texttt{)} ]\!] e_i]\ e_{i+1} \text{ in} \\
\quad \ldots \\
\quad \text{let } e_n = [[\![ id_n ]\!] \leftarrow \mathbf{EA}[\![ exp_n ]\!] e_{n-1}]\ e_{n-1} \text{ in} \\
\quad\quad e_n
\end{array}$$

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

$$\begin{array}{l}
\mathbf{T}[\![ \texttt{stop}\ exp_t\ \texttt{in}\ exp_r ]\!]\ =\ \lambda e. \\
\quad \text{case } \mathbf{T}[\![ exp_t ]\!] e \text{ of isListType}(T) \rightarrow \\
\quad \text{case } T \text{ of isTaskType}(S) \rightarrow \\
\quad\quad \mathbf{T}[\![ exp_r ]\!] e
\end{array}$$

$$\begin{array}{l}
\mathbf{EA}[\![ \texttt{stop}\ exp_t\ \texttt{in}\ exp_r ]\!]\ =\ \lambda e. \\
\quad \text{let} \\
\quad\quad (n, t) =\ \mathbf{EA}[\![ exp_t ]\!] e \\
\quad\quad c =\ \max(n, \text{maxcnt}(t)) \\
\quad \text{in } (c = 1) \rightarrow \mathbf{EA}[\![ exp_r ]\!] e
\end{array}$$

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 in the remaining of the paper not use **EA** any more but resort to the simple valuation **E**.

## 2.11 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

$$\mathbf{ES}: \quad \text{Expression} \to \mathbb{P}(\text{ValueEnv}) \to \mathbb{P}(\text{Value}) \qquad \text{Value Set}$$

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 [Sch93a]) 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:

$$\begin{aligned}
&\mathbf{T}[\![\texttt{any(}\mathit{exp}\texttt{)}]\!] = \underline{\lambda} e. \\
&\quad \text{case } \mathbf{T}[\![\mathit{exp}]\!]e \text{ of isListType}(T) \to \text{case } T \text{ of isTaskType}(S) \to \\
&\qquad \text{mkTupleType}(S, \text{mkListType}(\text{mkTaskType}(S)))
\end{aligned}$$

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

$$\begin{aligned}
&\text{any} : \text{List} \to \mathbb{P}(\text{Tuple}) \\
&\text{any}(t) = \\
&\quad \text{let } (n, f) = \text{seq}(t) \\
&\quad \text{in } \{\text{mkTuple}_2(s(0), \text{list}(s, 1, n)) \mid \exists s : \mathbb{N}^n \to \text{Value}, \exists p : \mathbb{N}^n \stackrel{\text{bij.}}{\to} \mathbb{N}^n, s(i) = f(p(i))\}
\end{aligned}$$

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 $\mathbb{N}^n \to$ 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:

$$
\begin{array}{l}
\mathrm{seq} : \mathrm{List} \rightarrow \mathbb{N} \times (\mathbb{N} \rightarrow \mathrm{Value}) \\
\mathrm{seq}(l) = \\
\quad \mathrm{case}\ l\ \mathrm{of} \\
\quad\quad \mathrm{isNil}() \ \rightarrow\ (0, \lambda x.\bot) \\
\quad\quad \mathrm{isList}(h,t) \ \rightarrow \\
\quad\quad\quad \mathrm{let}\ (n,f) =\ \mathrm{seq}(t) \\
\quad\quad\quad \mathrm{in}\ (n+1, \lambda x.(x = n) \rightarrow h; f(x))
\end{array}
$$

$$
\begin{array}{l}
\mathrm{list} : (\mathbb{N} \rightarrow \mathrm{Value}) \times \mathbb{N} \times \mathbb{N} \rightarrow \mathrm{List} \\
\mathrm{list}(f,i,n) = (i > n) \rightarrow \mathrm{mkNil}();\ \mathrm{mkList}(f(i), \mathrm{list}(f, i+1, n))
\end{array}
$$

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 $\mathbf{E}$ of the binary operator +

$$
\begin{array}{l}
\mathbf{E}[\![ exp_1\ \texttt{+}\ exp_2 ]\!] = \underline{\lambda} e. \\
\quad \mathrm{case}\ \mathbf{E}[\![ exp_1 ]\!] e\ \mathrm{of}\ \mathrm{isInt}(i_1) \rightarrow \\
\quad \mathrm{case}\ \mathbf{E}[\![ exp_2 ]\!] e\ \mathrm{of}\ \mathrm{isInt}(i_2) \rightarrow \\
\quad\quad \mathrm{mkInt}(i_1 + i_2)
\end{array}
$$

we get the embedded non-determinstic valuation **ES**

$$
\begin{array}{l}
\mathbf{ES}[\![ exp_1\ \texttt{+}\ exp_2 ]\!] = \underline{\lambda} E. \\
\quad \bigcup \{ \mathrm{sum}(v_1, v_2) | \exists e \in E, v_1 \in \mathbf{ES}[\![ exp_1 ]\!] e, v_2 \in \mathbf{ES}[\![ exp_2 ]\!] e \}
\end{array}
$$

where

$$
\begin{array}{l}
\mathrm{sum} : (\mathrm{Value} \times \mathrm{Value}) \rightarrow \mathrm{Value} \\
\mathrm{sum}(v_1, v_2) = \\
\quad \mathrm{case}\ v_1\ \mathrm{of}\ \mathrm{isInt}(i_1) \rightarrow \\
\quad \mathrm{case}\ v_2\ \mathrm{of}\ \mathrm{isInt}(i_2) \rightarrow \\
\quad\quad \mathrm{mkInt}(i_1 + i_2)
\end{array}
$$

For the other sorts of expressions, the transformation is basically the same. In the body of the definition all applications of $\mathbf{E}$ 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 1 (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* $[\![\mathtt{any}(exp)]\!]$ *is not deterministic.*

but also, by above construction,

**Lemma 2 (Embedding)** *All sorts of expressions defined in [Sch93a] are deterministic when embedded into the non-deterministic framework.*

We will apply Definition 1 later to a more interesting case.

## 2.12 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 sequential core language [Sch93a], cyclic definitions are forbidden. The order of definitions does not matter, but there has to exist some topolical 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 4.4 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 generating the original construction:

$$\mathbf{E}[\![\texttt{let}\ cdefs\ \texttt{in}\ exp]\!] = \underline{\lambda} e.\mathbf{E}[\![exp]\!](\mathbf{DE}(\mathbf{FD}[\![cdefs]\!])e)$$

$$\mathbf{FD}\left[\!\!\left[\begin{array}{l} \ldots \\ x\ \texttt{= forward;} \\ y\ \texttt{= fstart(}x\texttt{,}\ f\texttt{,}\ x\texttt{,}\ \ldots\texttt{);} \\ \ldots \\ \boxed{\ldots x \ldots} \end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l} \ldots \\ y\ \texttt{= start(}f\texttt{,}\ y\texttt{,}\ \ldots\texttt{);} \\ \ldots \\ \boxed{\ldots y \ldots} \end{array}\right]\!\!\right]$$

1. The `forward` definition is removed and every occurence 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).

$$\begin{array}{l} \mathbf{DE}[\![id_1^{T_1}\ \texttt{=}\ exp_1\ \texttt{;}\ \ldots\ \texttt{;}\ id_n^{T_n}\ \texttt{=}\ exp_n]\!] = \underline{\lambda} e. \\ \quad \mathrm{fix}(\lambda e'. \\ \qquad [[\![id_1]\!] \leftarrow \mathbf{E}[\![exp_1]\!]e' \\ \qquad \ldots \\ \qquad [\![id_n]\!] \leftarrow \mathbf{E}[\![exp_n]\!]e']e) \end{array}$$

## 2.13 Code Generation

The generation of C code for the parallel extensions of the core language is rather straightforward:

1. Tasks are represented by pointers to result structures as described in [HNS93]; 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 $\texttt{[ ]}_\texttt{S}$ and $\texttt{:}_\texttt{S}$ 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

   $$\mathbf{DC}\ [[\texttt{start(}f\texttt{,}\ a_1\texttt{,}\ \ldots\texttt{,}\ a_n\texttt{)}]] = [[\texttt{START}n\texttt{(}f\texttt{,}\ a_1\texttt{,}\ \ldots\texttt{,}\ a_n\texttt{)}]]$$

   $$\mathbf{DC}\ [[\texttt{wait(}task\texttt{)}]] = [[\texttt{WAIT(}task\texttt{)}]]$$

   $$\mathbf{DC}\ [[\texttt{any(}tlist\texttt{)}]] = [[\texttt{ANY(}tlist\texttt{)}]]$$

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 [Sch93a]):

$$\mathbf{DC}\left[\!\!\left[\boxed{\begin{array}{l}\ddots\\ \qquad\quad \texttt{stop}\ \mathit{tlist}\ \texttt{in}\ \mathit{exp}\end{array}}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\texttt{STOP(}\mathit{tlist}\texttt{);}\\ \boxed{\begin{array}{l}\ddots\\ \qquad\quad \mathit{exp}\end{array}}\end{array}\right]\!\!\right]$$

   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

$$\mathbf{DC}\ [[\texttt{fstart(}t\texttt{,}\ f\texttt{,}\ a_1\texttt{,}\ \ldots\texttt{,}\ a_n\texttt{)}]] = [[\texttt{FSTART}n\texttt{(}t\texttt{,}\ f\texttt{,}\ a_1\texttt{,}\ \ldots\texttt{,}\ a_n\texttt{)}]]$$

$$\mathbf{DC}\ [[\texttt{forward}]] = [[\texttt{FORWARD}]]$$

The external runtime interface described in Appendix B maps `START`, `WAIT`, `ANY`, `STOP`, `FORWARD`, `FSTART` (with minor modifications) into the corresponding PACLIB operations `pacVirtual`, `pacWaitOne`, `pacWaitList`, `pacStopList`, `pacForward` and `pacFStart`

Using `pacVirtual` instead of the conventional `pacStart` means that we apply by default the new PACLIB support for *virtual tasks* described in [Sch94]; consequently extremely many tasks with small granularity may be efficiently utilized.

# 3 Non-Deterministic Composition

## 3.1 Motivation

Section 2 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 synchronizions 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.

## 3.2 Non-Deterministic Composition

We introduce the new construct

$$\texttt{ndcomp(}f\texttt{,}\ b\texttt{,}\ p\texttt{)}\ \texttt{[}\ e\ \texttt{|}\ I_1\ \texttt{|@}par_1\ \ldots\ I_n\texttt{|@}par_n\ \texttt{]}$$

which is new a variant of *iterated compositions* [Sch93a]. Informally, 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. Moreover, optionally a predicate $p$ may be given with the property

$$p(x) \to p(f(x, y))$$

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 $V_1, V_2, \dots, V_n$. For every $V_i$ a task is started that sequentially evaluates $e$ in all bindings of that sequence and combines the resulting values to $A_i = f(a_n, f(a_{n-1}, \dots f(a_2, a_1)))$.

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. At last the return value of the last task is combined with the base value $b$ yielding the final result.

The boolean expression `|@`*par* may be 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. The token `|@` is interpreted as `|@True`; `|` denotes `|@False`.

Before giving a formal definition of the construct, we show some examples of its application.

### 3.3 Examples

The following function sums up all elements of a list; for every sublist with 100 elements a new task is started:

```
sum(x,y) = x+y
lsum(l) = ndcomp(sum, 0) [ x | x <- l, i <- 1.. |@ (i%100 == 0) ]
```

The next function 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 predicate does not hold the test may be aborted.

```
and(x,y) = x && y; not(x) = ^x
forall(l,p) = ndcomp(and, True, not) [ p(x) | x <- l |@ ]
```

We compute the product of all factors of all numbers in a list; for every element a task is started that factors this number and (sequentially) computes the corresponding subproduct.

```
prod(x,y) = x*y; factors(x) = ...
factprod(l) = ndcomp(prod, 1) [ y | x <- l |@ y <- factors(x) ]
```

### 3.4 Translation

We define the meaning of the non-deterministic composition construct by translating it into the parallel core language. This resembles the translation of iterated compositions (see [Sch93a]) but is considerably more complicated and raises several subtle questions.

In Figure 3 the top-level translation of the construct is shown. The result of the core translation is a tuple $r$ whose first component denotes whether the composition has yielded

$$
\mathbf{TC}\,[\![\ \texttt{ndcomp}(f,b,p)\ \texttt{[}\ e\ \texttt{|}\ I\ \texttt{]}\ ]\!] = \left[\!\!\left[\begin{array}{l}
\texttt{let} \\
\quad \texttt{v = } b \\
\quad \texttt{r = } \mathbf{TN}[\![I]\!][\![f]\!][\![e]\!][\![\texttt{v}]\!][\![p]\!] \\
\texttt{in if r.0} \\
\quad \texttt{then } f(r.\texttt{1, v)} \\
\quad \texttt{else v}
\end{array}\right]\!\!\right]
$$

Figure 3: Non-deterministic Composition (Top-Level)

$\mathbf{TN}[\![\ ]\!][\![f]\!][\![e]\!][\![v]\!][\![p]\!]\ =\ [\![\ \texttt{(True, } e\texttt{)}\ ]\!]$

$\mathbf{TN}[\![h_i\texttt{:}t_i\texttt{<-}l_i\texttt{,}\ x_j\texttt{<-}a_j\texttt{..}b_j\texttt{+=}c_j\texttt{,}\ y_k\texttt{=}u_k\texttt{,}\ defs_1\texttt{;}\ s\texttt{;}\ y_k\texttt{'}\ \texttt{=}\ r_k\texttt{,}\ defs_2\ \texttt{|@}par\ R]\!]$
$[\![f]\!][\![e]\!][\![v]\!][\![p]\!]\ =$

```
[[ let
     auxiliary definitions
     G(hᵢ, tᵢ, xⱼ, yₖ) =
       let defs_1; defs_2
       in TN[[R]][[f]][[e]][[v]][[p]]
     comb(zsᵢ, xsⱼ, ysₖ, n) =
       loop( (False, v ) ) [ sum |
         hᵢ:tᵢ <- zsᵢ, xⱼ <- xsⱼ .. += cⱼ, yₖ <- rev(zsₖ), i <- 1..n,
         sum = (False, v), new = G(hᵢ, tᵢ, xⱼ, yₖ),
         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(zᵢ, xⱼ, yₖ, zsᵢ, xsⱼ, ysₖ, n, bag) =
       case zᵢ of
         [ ] => red(start(comb, zsᵢ, xsⱼ, ysₖ, n):bag)
         hᵢ:tᵢ =>
           if xⱼ > bvⱼ
             then red(start(comb, zsᵢ, xsⱼ, ysₖ, n):bag)
             else let
               defs_1
             in if ^s
               then red(start(comb, zsᵢ, xsⱼ, ysₖ, n):bag)
               else let
                 defs_2
               in if par
                 then F(tᵢ, xⱼ+cvⱼ, rₖ, zᵢ, xⱼ, [yₖ], 1,
                   start(comb, zsᵢ, xsⱼ, ysₖ, n):bag)
                 else F(tᵢ, xⱼ+cvⱼ, rₖ, zsᵢ, xsⱼ, yₖ:ysₖ, n+1, bag)
     lvⱼ = lⱼ; avⱼ = aⱼ; bvⱼ = bⱼ; cvⱼ = cⱼ
   in
     F(lvᵢ, avⱼ, uₖ, lvᵢ, avⱼ, [ ], 0, [ ])                              ]]
```

Figure 4: Non-deterministic Composition (Core)

a result (the iteration range may have been empty); if this is not the case, the result of the construct is the base value $b$. Otherwise, the second tuple component holds the computed value that is combined with $b$ to yield the final result.

Figure 4 shows the actual core of the translation:

1. Each iterator $I$ is transformed into a recursive function $F$ (which is tail-recursive and therefore compiled 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 bag 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`.

   (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 iteration constructs into the recursive function 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 [Sch93a]). Here 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 [Sch93a]) is actually not required and might be replaced by low-level applications of `head` and `tail` only (this is not yet done).

```
rev(l) = rlist [ x | x <- l ]

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 5: Non-Deterministic Composition (Auxiliaries)

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).

Figure 5 shows several auxiliary functions that are required in above translation but that are (mainly) 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` 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.

The dynamic behavior of the construct is illustrated in Figure 6. The evaluation of $e$, the combination $f$ and all definitions only used within the remaining iterators $I$ and $e$ are not executed by the current task but by the created subtasks. However, all computations

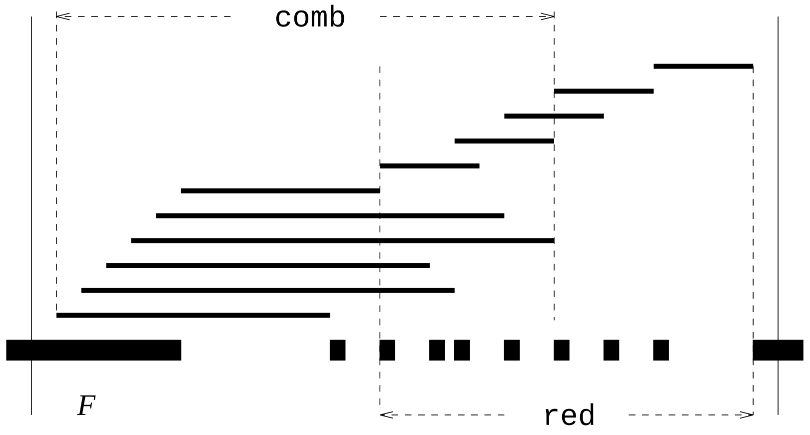


Figure 6: Non-deterministic Composition (Dynamical Behavior)

required to construct the sequence of variable bindings are performed sequentially by the main task. The rate at which tasks can be started therefore depends on the complexity of the "sequential path". Afterwards, the computation of the individual values may overlap with the reduction to a single result.

### 3.5 Restricted Non-Determinism

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 3 (Non-Deterministic Composition)** *Let $f : T \times T \to T$ be a commutative and associative operation over $T$ and $p : T \to \mathrm{Bool}$ be a predicate over $T$ such that*

$$p(x) \to p(f(x, y))$$

*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 orderered with maximum $m$, and $f : T \times T \to T$ 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 $T = \mathrm{Bool}$ with true $\sqsubseteq$ false in the function `forall` in Subsection 3.3).

A formal proof of Lemma 3 using the denotational framework would become rather lengthy and not provide much further insight. We therefore comit to the following

**Proof Sketch:** We define the denotation of non-deterministic compositions as

$$\mathbf{ES}[\![\texttt{ndcomp(}f\texttt{, }b\texttt{, }p\texttt{) [ }e\texttt{ | }Is\texttt{]}]\!]S := \mathbf{ES}\ (\mathbf{TC}[\![\texttt{ndcomp(}f\texttt{, }b\texttt{, }p\texttt{) [ }e\texttt{ | }Is\texttt{]}]\!])S$$

Let $S$ be a singleton environment. The denotation

$$\mathbf{ES}[\![\texttt{ndcomp(}f\texttt{, }b\texttt{, }p\texttt{) [ }e\texttt{ | ]}]\!]S$$

is certainly a singleton. Now consider

$$\mathbf{ES}[\![\texttt{ndcomp(}f\texttt{, }b\texttt{, }p\texttt{) [ }e\texttt{ | }I\texttt{ |@}par\ Is\texttt{ ]}]\!]S$$

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 $g : \mathbb{N}^m \rightarrow \mathbb{N}^n$ 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 $p$(`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 2:

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.

## 3.6 Possible Optimizations and Extensions

Our translation scheme yields a program that traverses the iteration range twice; once by the main task and once by the generated subtasks. In general this is necessary since the variable bindings are generated in sequence. However, in special cases several optimizations are possible (but are 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 of the translation scheme 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.

Furthermore, `nondet` is currently more restrictive than the other composition constructs in 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($f'$, $b$) [ $g(e)$ | $ls$]

by the programmer.

# 4 Para-Functional Annotations

## 4.1 Motivation

The parallel core language defined in Section 2 serves as a "pseudo-functional" 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 efficent 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 another scheme where the programmer uses type-transparent annotations to trigger parallel computations. From these annotations the compiler generates the necessary `start` calls, analyses the program, and inserts the required `wait` invokations. The goal to maximize exploitable parallelism but minimize the synchronization overhead. The resulting program is strongly typed and can be compiled without any 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.

| | | | |
|---|---|---|---|
| $exp$ | $\rightarrow$ | ... | Basic Expressions |
| | \| | $exp$`@` | Task Annotation |
| | \| | $exp_h$ `:@` $exp_t$ | Stream Annotation |
| | \| | $exp$`!` | Synchronization Annotation |
| $defs$ | $\rightarrow$ | ... | Basic Definitions |
| | \| | `para` $id$ `::` $type$ | Para-Type Signature |

Figure 7: Para-Functional Annotations

## 4.2 The Annotation Language

In Figure 7 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 the type checking process. 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 enrichen 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).

## 4.3 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.

## 4.4 Semantics

In the following paragraphs we give a denotational description of the annotations that is partially still 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 Subsection 4.5.

The following definitions use the function

$$
\begin{array}{l}
\mathrm{sync} : \mathrm{Value} \rightarrow \mathrm{Value} \\
\mathrm{sync}(e) = \\
\quad \mathrm{case}\ e\ \mathrm{of} \\
\qquad \mathrm{isBool}(b) \rightarrow b \\
\qquad \mathrm{isInt}(i) \rightarrow i \\
\qquad \mathrm{isList}(h,t) \rightarrow \mathrm{mkList}(\mathrm{sync}(h),\ \mathrm{sync}(t)) \\
\qquad \ldots \\
\qquad \mathrm{isTask}(t) \rightarrow \mathrm{sync}(t()) \\
\qquad \mathrm{isStream}(h,t) \rightarrow \mathrm{mkList}(\mathrm{sync}(h), \mathrm{sync}(t))
\end{array}
$$

that synchronizes any 'Value' into a ground object that does not contain any 'Task' (and consequently also not any 'Stream' object).

### Task Annotations

$$\mathbf{TE}[\![ exp\texttt{@} ]\!] = \underline{\lambda} e.\mathrm{mkTask}(\underline{\lambda} v.\mathbf{TE}[\![ exp ]\!] e)$$

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.

$$\mathbf{TE}[\![ \hat{}\, exp ]\!] = \underline{\lambda} e.\mathrm{sync}(\mathbf{TE}[\![ exp ]\!] e)$$

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*.

### Stream Annotations

$$\begin{aligned}&\mathbf{TE}[\![ exp_h \,\texttt{:@}\; exp_t ]\!] = \underline{\lambda} e.\mathrm{mkStream}(a, b) \text{ where}\\ &\quad a = adapt[T](\mathbf{TE}[\![ exp_h ]\!] e)\\ &\quad b = \mathrm{mkTask}(\underline{\lambda} v.\, adapt[\mathrm{Stream}(T)](\mathbf{TE}[\![ exp_t ]\!] e))\end{aligned}$$

$exp_t$ 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 guarantuee 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 $T$.

### Synchronization Annotations

$$\mathbf{TE}[\![ exp\texttt{!} ]\!] = \underline{\lambda} e.\mathrm{sync}(\mathbf{TE}[\![ exp\texttt{!} ]\!] e)$$

The annotation guarantuees 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.

**Para-Type Signatures**

$$
\begin{array}{l}
\mathbf{DE}[\![ \ldots \texttt{ para } id :: type;\ id = exp\ \ldots ]\!] = \underline{\lambda} e. \\
\quad \mathrm{fix}(\lambda e'. \\
\qquad [\ldots \\
\qquad\quad [\![ id ]\!] \leftarrow extend[type](\mathbf{E}[\![ exp ]\!] e') \\
\qquad \ldots] e)
\end{array}
$$

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 applied to an optimization described in [Sch93a]).

**Cyclic Constant Definitions**

The para-annotated language allows to write cyclic constant definitions as described in Subsection 2.12. 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 2.12 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 annoated 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@)@
   ```

$$\text{Task}\ \frac{T \rightarrow_{\mathrm{T}} U}{T \rightarrow_{\mathrm{T}} \mathrm{Task}(U)} \qquad \text{Tuple}\ \frac{T_1 \rightarrow_{\mathrm{T}} U_1, \ldots, T_n \rightarrow_{\mathrm{T}} U_n}{(T_1, \ldots, T_n) \rightarrow_{\mathrm{T}} (U_1, \ldots, U_n)}$$

$$\text{List}\ \frac{T \rightarrow_{\mathrm{T}} U}{[T] \rightarrow_{\mathrm{T}} [U]} \qquad \text{Stream}\ \frac{T \rightarrow_{\mathrm{T}} U}{[T] \rightarrow_{\mathrm{T}} \mathrm{Stream}(U)}$$

Figure 8: A Type Relation

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 4.10 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.

## 4.5 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 2 (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*
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 3 (Type Order)** *The binary type relation $\rightarrow_{\mathrm{T}}$ is the reflexive transitive closure of the derivation rules given in Figure 8: $T \rightarrow_{\mathrm{T}} T'$ holds if and only if there is a sequence of $n \geq 0$ derivations $T = T_0 \rightarrow_{\mathrm{T}} T_1 \rightarrow_{\mathrm{T}} \ldots \rightarrow_{\mathrm{T}} T_n = T'$.*

We use the same symbol $\rightarrow_{\mathrm{T}}$ 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 4 (Synchronization/Extension)** *If $T \rightarrow_{\mathrm{T}} U$, there exists an extension function*

$$\mathrm{ext}[T, U] = e : T \rightarrow U$$

*and acorresponding 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 4 (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*

$$\begin{array}{l} T \rightharpoondown_{\mathrm{T}} T_1, T \rightharpoondown_{\mathrm{T}} T_2, \\ \forall U.(U \rightharpoondown_{\mathrm{T}} T_1, U \rightharpoondown_{\mathrm{T}} T_2) \to T \not\rightharpoondown_{\mathrm{T}} U \end{array}$$

*(if such a $T$ exists).*

**Definition 5 (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*

$$\begin{array}{l} T_1 \rightharpoondown_{\mathrm{T}} T, T_2 \rightharpoondown_{\mathrm{T}} T, \\ \forall U.(T_1 \rightharpoondown_{\mathrm{T}} U, T_2 \rightharpoondown_{\mathrm{T}} U) \to U \not\rightharpoondown_{\mathrm{T}} T \end{array}$$

*(if such a $T$ exists).*

**Lemma 5 (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 $T_1 \rightharpoondown_{\mathrm{T}} T_2$ 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

$$\begin{array}{ccc} T_1 & & T_2 \\ \uparrow & \times & \uparrow \\ U_1 & & U_2 \end{array}$$

We may assume that $T_1 \not\rightharpoondown_{\mathrm{T}} T_2$ (and $T_2 \not\rightharpoondown_{\mathrm{T}} T_1$) since otherwise $U_1 = U_2 = T_1$ (respectively $U_1 = U_2 = T_2$).

We may also assume that in $U_1 \rightharpoondown_{\mathrm{T}} \ldots \rightharpoondown_{\mathrm{T}} T' \rightharpoondown_{\mathrm{T}} T_1$ and $U_2 \rightharpoondown_{\mathrm{T}} \ldots \rightharpoondown_{\mathrm{T}} S' \rightharpoondown_{\mathrm{T}} T_1$ 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 $T_1 = \text{Stream}(T)$ we have a contradiction $T' = S' = [T]$.
3. If $T_1 = [a]$, then $U_1 = [b]$ with $b \rightarrow_{\text{T}} a$ and $U_2 = [c]$ with $c \rightarrow_{\text{T}} a$. By induction we may construct a unique $V = [b \sqcup_{\text{T}} c]$. By the definition of $\rightarrow_{\text{T}}$ we know that $T_2 = \text{Task}^j([d])$ or $T_2 = \text{Task}^j(\text{Stream}(d))$ for some $d$ and $j \geq 0$. By definition of $\sqcup_{\text{T}}$ we thus have $V \rightarrow_{\text{T}} T_2$ but also $V \rightarrow_{\text{T}} T_1$. By assumption (no common types in the derivation sequence) we know that $T_1 \rightarrow_{\text{T}} V$ and consequently $T_1 = V$. This yields the contradiction $T_1 \rightarrow_{\text{T}} T_2$.
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 $b_i \rightarrow_{\text{T}} a_i$ and $c_i \rightarrow_{\text{T}} a_i$ for all $i$. By induction we may construct a unique $V = (b_1 \sqcup_{\text{T}} c_1, \ldots, b_n \sqcup_{\text{T}} c_n)$. By definition of $\rightarrow_{\text{T}}$ we know that $T_2 = \text{Task}^j((d_1, \ldots, d_n))$ for some $d_i$ and $j \geq 0$. By definition of $\sqcup_{\text{T}}$ we have $V \rightarrow_{\text{T}} T_2$ but also $V \rightarrow_{\text{T}} T_1$. By assumption (no common types in the derivation sequence) we know that $T_1 \rightarrow_{\text{T}} V$ and consequently $T_1 = V$. This yields the contradiction $T_1 \rightarrow_{\text{T}} T_2$.

The induction step for $T_1 \sqcup_{\text{T}} T_2$ 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 guarantuee termination of the algorithm, it is necessary to introduce the following

**Definition 6 (Maximum Extension Type)** *The maximum extension* $\max_{\text{T}}(T, T')$ *of a type* $T$ *in direction* $T'$ *(with* $T \rightarrow_{\text{T}} T'$*) is the largest type* $U$ *with* $T \rightarrow_{\text{T}} U \rightarrow_{\text{T}} T'$ *such that* $U$ *does not contain any subtype* $\text{Task}(\text{Task}(V))$*. 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 6 (Uniqueness)** *For every pair of types* $T \rightarrow_{\text{T}} T'$, $\max_{\text{T}}(T, T')$ *exists and is unique.*

**Proof:** We proceed by induction over the structure of $T$. If $T$ is atomic, then either $T' = T = \max_{\text{T}}(T, T')$ or $T' = \text{Task}^i(T)$ and $\max_{\text{T}}(T, T') = \text{Task}(T)$.

In case $T = [U]$ we have $T' = \text{Task}^i(\text{List}(U'))$ (or $T' = \text{Task}^i(\text{Stream}(U'))$) for some $U \rightarrow_{\text{T}} U'$ and $i \geq 0$. Let $V = \max_{\text{T}}(U, U')$.

If $V$ contains a subtype with two tasks, then

$$\max_{\text{T}}(T) = [V] \text{ (respectively } \max_{\text{T}}(T) = \text{Stream}(V))$$

Otherwise

$$\max_{\text{T}}(T) = \text{Task}([V]) \text{ (respectively } \max_{\text{T}}(T) = \text{Task}(\text{Stream}(V)))$$

The other cases proceed in a similar fashion. $\square$

Finally we need the following

**Definition 7 (Maximum Extension Set)** *The maximum extension* $\max_{\text{T}}(T)$ *of a type* $T$ *is*

$$\{\max_{\text{T}}(T, T') | T \rightarrow_{\text{T}} T'\}$$

The transformation algorithm described in Section 4.6 requires this upper bound for the types of function arguments/results to guarantuee termination.

**Lemma 7 (Finiteness)** *For every* $T$, $\max_{\text{T}}(T)$ *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 $\max_{\text{T}}(T, T') = T_m$ for all $T'$ with $T_m \rightarrow_{\text{T}} T'$. The union of the sets all types $T \rightarrow_{\text{T}} \ldots \rightarrow_{\text{T}} T_m$ in all paths is finite and contains all maximum extension types. $\square$

## 4.6 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

   $$\begin{array}{l} f : S \rightarrow T \\ f(x) = e \end{array}$$

   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

   $$\begin{array}{l} f : S \rightarrow T; f(x) = \ sync[T', T](f'(x)) \\ f' : S \rightarrow T'; f'(x) = \ sync[T'', T'](e'') \end{array}$$

   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.** $ext[T, T']$ (with $T \rightarrow_{\mathrm{T}} T'$) is the extension function $e : T \rightarrow T'$.

2. **Synchronization.** $sync[T', T]$ (with $T \rightarrow_{\mathrm{T}} T'$) is the synchronization function $s : T' \rightarrow T$.

3. **Synchronized Constants.** $syncval[T', T, c']$ (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

   $$\begin{array}{l} syncret[T', T, c'] := c \\ c = sync[T', T](c') \\ c' = sync[T'', T'](e'') \end{array}$$

   where $T'' = \max_{\mathrm{T}}(T, T')$ i.e. $T \rightarrow_{\mathrm{T}} T' \rightarrow_{\mathrm{T}} T''$. Every further occurence of $c$ is replaced by an occurence of $c'$.

4. **Synchronized Results.** $syncret[T', T, f']$ (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

   $$\begin{array}{l} syncret[T', T, f'] := f \\ f(x) = sync[T', T](f'(x)) \\ f'(x) = sync[T'', T'](e'') \end{array}$$

   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.** $syncarg[T', T, f]$ (with $T \to_{\mathrm{T}} T'$) is a function $f$ with some application $f(a)$ whose argument type has been extended from $T$ to $T'$.

   $$syncarg[T', T, f'] := f$$
   $$f'(x) = f(\mathrm{sync}[T', T](x))$$

   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 4.7.

   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

   $$syncret[S', S, f'''] := f''$$
   $$f''(x) = sync[S', S](f'''(x))$$
   $$f'''(x) = f'(sync[T'', T'](x)))$$

   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 8 (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 9 (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 10, 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 11 and 12, simplification is correct and terminates.
3. **Argument Extension:** By Lemma 13 and 14, the transformation of the expression $syncarg[T', T, f](x)$ does not change the result and terminates.
4. **Result Extension:** By Lemma 15 and 15, 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 4.7) 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 4.7), we have $U \in \max_{\mathrm{T}}(T)$ for every $f' : S \rightarrow U$ with $f = syncret[U, T, f']$.

By Lemma 7 (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". □

## 4.7 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 10 (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 occurence 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,

$$\frac{\begin{array}{c}\ldots\\ c = \mathrm{syncval}[T', T, c']\end{array}}{\ldots}$$

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

$$\frac{\ldots}{\begin{array}{c}\ldots\\ c = \mathit{syncval}[T', T, c']\end{array}}$$

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 does 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 guarantuees that any rule that becomes applicable by an extension of the environment is eventually applied.

### Task Annotations

A para-annotated expression $e$ (with free variables $x_1, \ldots, x_n$) is abstracted into a function $f = \underline{\lambda} x_1 \ldots \underline{\lambda} x_n . e$ from which a `start` expression is constructed:

$$\frac{e \vdash d : T}{e\texttt{@} \;\vdash\; (\texttt{let } f\texttt{(}x_1\texttt{, ..., }x_n\texttt{) = }d\texttt{ in start(}f\texttt{, }x_1\texttt{, ..., }x_n\texttt{))} : \mathit{Task}(T)}$$

### 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 $\mathrm{Task}(\mathrm{Stream}(T))$.

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):

$$\frac{\begin{array}{c} e_h \vdash d_h : T_h, e_t \vdash d_t : T_t \\ T_t = \mathrm{Task}^i(S), S = [U], V = T_h \sqcup_{\mathrm{T}} U, W = \mathrm{Stream}(V) \end{array}}{e_h \;\texttt{:@}\; e_t \;\vdash\; (\mathrm{ext}[T_h, V](d_h) \;\texttt{:s}\; \mathrm{ext}[T_t, \mathrm{Task}(W)](d_t)) : W}$$

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.

$$\frac{e_h \vdash d_h : T_h, e_t \vdash d_t : T_t \quad T_t = \mathrm{Task}^i(S), S = \mathrm{Stream}(U), V = T_h \sqcap_{\mathrm{T}} U}{e_h \texttt{ :@ } e_t \ \vdash \ (\mathrm{ext}(V, U)(\mathrm{sync}[T_h, V](d_h)) \texttt{ :s } \mathrm{ext}[S, \mathrm{Task}(S)](\texttt{sync}[T_t, S](d_t))) : S}$$

### 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).

$$\frac{e \vdash d : T \quad U = \mathrm{ground}(T)}{e\texttt{!} \ \vdash \ \mathrm{sync}[T, U](d) : U}$$

### 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.

$$\frac{e \vdash d : T, a \vdash b : B \quad T \rightarrow_{\mathrm{T}} T', U = \max_{\mathrm{T}}(S, T')}{\begin{array}{c}\texttt{let para } c \texttt{ :: } T'\texttt{; } c^S \texttt{ = } e \texttt{ in } a \ \vdash \\ (\texttt{let } c'^U \texttt{ = } \mathit{sync}[T', U](\mathit{ext}[T, T'](d))\texttt{, } c^S \texttt{ = } \mathrm{sync}[U, S](c') \texttt{ in } b) : B \\ \mathit{syncval}\,[U, S, c'] = c\end{array}}$$

### 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 $c'$.

$$\frac{e \vdash d : T, a \vdash b : B \quad U = \max_{\mathrm{T}}(S, T)}{\begin{array}{c}\texttt{let } c^S \texttt{ = } e \texttt{ in } a \ \vdash \ (\texttt{let } c'^U \texttt{ = } \mathit{sync}[T, U](d)\texttt{, } c^S \texttt{ = } \mathrm{sync}[U, S](c') \texttt{ in } b) : B \\ \mathit{syncval}[U, S, c'] = c\end{array}}$$

#### 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 4.11).

$$\frac{c = \mathit{syncval}[U, T, c']}{c \vdash\ c' : U}$$

#### 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 $f'$.

$$\frac{\begin{array}{c} e \vdash d : T, a \vdash b : B \\ U = \ \max_{\mathrm{T}}(S, T) \end{array}}{\begin{array}{c} \texttt{let } f^{R \to S}(x_j) \texttt{ = } e \texttt{ in } a\ \vdash \\ (\texttt{let } f'^{R \to U}(x) \texttt{ = } \mathit{sync}[T, U](d)\texttt{, } f^{R \to S}(x) \texttt{ = } \mathrm{sync}[U, S](f'(x)) \texttt{ in } b) : B \\ \mathit{syncret}[U, S, f'] = f \end{array}}$$

#### 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:

$$\frac{f = \mathit{syncret}[T, S, f']}{f\texttt{(}e\texttt{)}\ \vdash\ f'\texttt{(}e\texttt{)} : T}$$

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).

$$\frac{\begin{array}{c} e \vdash d : T \\ U = \max_{\mathrm{T}}(S, T) \end{array}}{\begin{array}{c} f^{S \to R}\texttt{(}e\texttt{)}\ \vdash\ \texttt{let } f'^{U \to R}\texttt{(}x\texttt{) = } f\texttt{(}\mathit{sync}[U, S](x)\texttt{) in } f'\texttt{(}\mathit{sync}[T, U](d)\texttt{)} : R \\ \mathit{syncarg}[U, S, f'] = f \end{array}}$$

#### 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.

$$\frac{e_c \vdash d_c : T_c, e_t \vdash d_t : T_t, e_f \vdash d_f : T_f \quad T = T_t \sqcap_{\mathrm{T}} T_f}{\texttt{if } e_c \texttt{ then } e_t \texttt{ else } e_f \vdash (\texttt{if } \mathrm{sync}[T_c, \mathrm{Bool}](d_c) \texttt{ then } \mathrm{sync}[T_t, T](d_t) \texttt{ else } \mathrm{sync}[T_t, T](d_f)) : T}$$

### 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.

$$\frac{e \vdash d : T \quad T = \mathrm{Task}^i(S), S = [U] \vee S = \mathrm{Stream}(U)}{\texttt{isnil}(e) \vdash \texttt{isnil}(\mathrm{sync}[T, S](d)) : \mathrm{Bool}}$$

$$\frac{e \vdash d : T \quad T = \mathrm{Task}^i(S), S = [U] \vee S = \mathrm{Stream}(U)}{\texttt{head}(e) \vdash \texttt{head}(\mathrm{sync}[T, S](d)) : U}$$

$$\frac{e \vdash d : T \quad T = \mathrm{Task}^i(S), S = [U]}{\texttt{tail}(e) \vdash \texttt{tail}(\mathrm{sync}[T, S](d)) : S}$$

$$\frac{e \vdash d : T \quad T = \mathrm{Task}^i(S), S = \mathrm{Stream}(U)}{\texttt{tail}(e) \vdash \texttt{tail}(\mathrm{sync}[T, S](d)) : \mathrm{Task}(S)}$$

### 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:

$$\frac{e_h \vdash d_h : T_h, e_t \vdash d_t : T_t \quad T_t = \mathrm{Task}^i(S), S = [U], V = T_h \sqcup_{\mathrm{T}} U}{e_h : e_t \vdash (\mathrm{ext}[T_h, V](d_h) : \mathrm{ext}[T_t, [V]](d_t)) : [V]}$$

Otherwise, the head and the tail are synchronized to the maximum common subtype:

$$\frac{e_h \vdash d_h : T_h, e_t \vdash d_t : T_t \quad T_t = \mathrm{Task}^i(S), S = [U], V = T_h \sqcap_{\mathrm{T}} U}{e_h : e_t \vdash (\mathrm{sync}[T_h, V](d_h) : \mathrm{sync}[T_t, [V]](d_t)) : [V]}$$

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.

$$\frac{e_h \vdash d_h : T_h, e_t \vdash d_t : T_t \quad T_t = \mathrm{Task}^i(S), S = \mathrm{Stream}(U), V = T_h \sqcap_{\mathrm{T}} U}{e_h : e_t \vdash (\mathrm{ext}(V, U)(\mathrm{sync}[T_h, V](d_h)) :_{\mathtt{S}} \mathrm{ext}[S, \mathrm{Task}(S)](\mathrm{sync}[T_t, S](d_t))) : S}$$

### 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.

$$\frac{e_h \vdash d_h : T_h, e_t \vdash d_t : T_t \quad T_t = \mathrm{Task}^i(S), S = \mathrm{Stream}(U), V = T_h \sqcap_{\mathrm{T}} U}{e_h \;\texttt{:s}\; e_t \;\vdash\; (\mathrm{ext}(V,U)(\mathrm{sync}[T_h,V](d_h)) \;\texttt{:s}\; \mathrm{ext}[S,\mathrm{Task}(S)](\mathrm{sync}[T_t,S](d_t))) : S}$$

### Tuple Selection

The basis expression must be synchronized to a tuple type.

$$\frac{e \vdash d : T \quad T = \mathrm{Task}^i(S), S = (S_1, \ldots, S_i, \ldots, S_n)}{e\texttt{.}i \;\vdash\; (\mathrm{sync}[T,S](d)\texttt{.}i) : S_i}$$

### Tuple Construction

The types of the tuple components are not constrained at all:

$$\frac{e_1 \vdash d_1 : T_1, \ldots, e_n \vdash d_n : T_n \quad T = (T_1, \ldots, T_n)}{\texttt{(}e_1\texttt{,}\ \ldots\texttt{,}\ e_n\texttt{)} \;\vdash\; \texttt{(}d_1\texttt{,}\ \ldots\texttt{,}\ d_n\texttt{)} : T}$$

### Unary and Binary Expressions

The arguments of strict (arithmetic and logical) operators are synchronized to the appropriate ground type. Two examples are given below:

$$\frac{e_1 \vdash d_1 : T_1, e_n \vdash d_n : T_n}{e_1 \;\texttt{+}\; e_2 \;\vdash\; (\mathrm{sync}[T_1,\mathrm{Int}](d_1) \;\texttt{+}\; \mathrm{sync}[T_2,\mathrm{Int}](d_2)) : \mathrm{Int}}$$

$$\frac{e \vdash d : T}{\texttt{\^{}}\; e \;\vdash\; (\texttt{\^{}}\; \mathrm{sync}[T,\mathrm{Bool}](d)) : \mathrm{Bool}}$$

### Constants

$$\frac{}{\mathit{intconst} \;\vdash\; \mathit{intconst} : \mathrm{Int}} \qquad \frac{}{\mathit{boolconst} \;\vdash\; \mathit{boolconst} : \mathrm{Bool}}$$

$$\frac{}{\texttt{[ ]} \;\vdash\; \texttt{[ ]} : \mathrm{Nil}} \qquad \frac{}{\texttt{[ ]}_{\texttt{S}} \;\vdash\; \texttt{[ ]}_{\texttt{S}} : \mathrm{SNil}}$$

### 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 nevery synchronized before the task is started; instead an auxiliary function takes the free variables of the arguments and performs the synchronization within the function.

$$\frac{e \vdash d : T \quad U = \max_{\mathrm{T}}(S,T)}{\begin{array}{l}\texttt{start(}fS \to R\texttt{,}\ e\texttt{)} \;\vdash\; \texttt{let}\ f'^{U \to R}\texttt{(}x\texttt{) = }f\texttt{(}\mathit{sync}[U,S](x)\texttt{);} \\ \qquad\qquad g^{X_j \to R}\texttt{(}x_j\texttt{) = }f'\texttt{(}\mathit{sync}[T,U](e)\texttt{)} \\ \qquad \texttt{in start(}g\texttt{,}\ x_j\texttt{)} : R\end{array}}$$

$$\mathit{syncarg}[U,S,f'] = f$$

### Task Synchronization

By a `wait` expression one 'Task' level is removed from an expression type:

$$\frac{e \vdash d : T \qquad T = \mathrm{Task}(S)}{\texttt{wait}(e) \ \vdash \ \texttt{wait}(d) : S}$$

### Non-Deterministic Synchronization

The task list argument of `any` has to be appropriately synchronized:

$$\frac{e \vdash d : T \qquad T = \mathrm{Task}^i(\mathrm{List}(S)) \vee T = \mathrm{Task}^i(\mathrm{Stream}(S))}{\texttt{any}(e) \ \vdash \ \texttt{any}(\mathrm{sync}[T, \mathrm{List}(S)](d)) : (S, \mathrm{List}(S))}$$

### Task Abortion

The task list argument of `stop` (but not its result) has to be appropriately synchronized:

$$\frac{e_t \vdash d_t : T_t, e_r \vdash d_r : T_r \qquad T_t = \mathrm{Task}^i(\mathrm{List}(S)) \vee T_t = \mathrm{Task}^i(\mathrm{Stream}(S))}{\texttt{stop}\ e_t\ \texttt{in}\ e_r \ \vdash \ (\texttt{stop}\ \mathrm{sync}[T_t, \mathrm{List}(S)](d_t)\ \texttt{in}\ d_r) : T_r}$$

## 4.8 Simplification

In the "Simplification" phase of the algorithm, nested applications of *sync* and/or *ext* applications are reduced as denoted by the following rules:

$$\frac{ext[T,T](e)}{e} \qquad \frac{sync[T,T](e)}{e}$$

$$\frac{ext[T',T''](ext[T,T'](e))}{ext[T,T''](e)} \qquad \frac{sync[T',T](sync[T'',T'](e))}{sync[T'',T](e)}$$

$$\frac{ext[T,T'](sync[T'',T](e)) \qquad T'' \rightarrow_{\mathrm{T}} T'}{ext[T'',T'](e)} \qquad \frac{ext[T',T](sync[T'',T'](e)) \qquad T \rightarrow_{\mathrm{T}} T''}{sync[T'',T](e)}$$

$$\frac{sync[T',T''](ext[T,T'](e)) \qquad T \rightarrow_{\mathrm{T}} T''}{ext[T,T''](e)} \qquad \frac{sync[T',T](ext[T'',T'](e)) \qquad T \rightarrow_{\mathrm{T}} T''}{sync[T'',T](e)}$$

**Lemma 11 (Correctness)** *The application of the simplification rules to an expression does not change the result.*

**Lemma 12 (Termination)** *The application of the simplification rules terminates.*

**Proof:** Correctness is a direct consequence Lemma 4. Since the application of a rule removes one level of *sync*/*ext* nestings, the algorithm terminates. □

## 4.9 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 13 (Correctness)** *Argument extension of a function $f$ does neither change the type interface nor the result of $f$.*

**Proof:** By Lemma 10. □

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 synchroniztion 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 $f : T \rightarrow S$ and a type $T''$ and returns the maximum $T'$ with $T \rightarrow_{\mathrm{T}} T' \rightarrow_{\mathrm{T}} T''$ which it "makes sense" to pass to $f$ without synchronization.

**Algorithm 2 (Maximum Argument Type)**

1. **Extension Space:** Construct the type space

   $$\{T' | T \rightarrow_{\mathrm{T}} T' \rightarrow_{\mathrm{T}} \max_{\mathrm{T}}(T, T'')\}$$

   and sort it in an topological order with respect to $\rightarrow_{\mathrm{T}}$.

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 $\overline{\text{List}}$ constructurs by 'List' (and $\overline{\text{Stream}}$ by 'Stream').

3. **Success:** If no `wait` is introduced by additional *sync* applications, the test suceeds; in this case return $T$.

4. **Failure:** Otherwise, remove all $U'$ with $U \rightarrow_{\mathrm{T}} U'$ 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 intution sketched above pretty well. Anyway, the programmer may use the annotation *exp*`!` to explicitly demand synchronization of values before the function is called.

**Lemma 14 (Termination)** *The algorithm "Maximum Argument Type" terminates.*

**Proof:** By Lemma 7 (Finiteness) the type space is finite, therefore the outer loop terminates. Furthermore, for every function $f : S \rightarrow T$, the set $\max_{\mathrm{T}}(S)$ is finite and so is the depth of the stack used by the inner loop. □

## 4.10 Result Extension

The goal of "Result Extension" is to remove those applications of $ext[T, T'](e)$ (with $T \rightarrow_{\mathrm{T}} T'$) 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 4.8 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 $T' = \text{Task}(T)$, $e$ is abstracted and a `start` call is created:

$$\frac{\begin{array}{c} ext[T,U](e) \vdash d \\ T' = \text{Task}(U) \end{array}}{ext[T,T'](e) \ \vdash \ \texttt{let } f(x) \texttt{ = } d \texttt{ in start(} f\texttt{,} x\texttt{)}}$$

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:

$$\frac{T = [U], T' = [U']}{ext[T,T'](\texttt{[ ]}) \ \vdash \ \texttt{[ ]}}$$

$$\frac{\begin{array}{c} ext[U,U'](e_h) \ \vdash d_h, \ ext[T,T'](e_t) \ \vdash d_t \\ T = [U], T' = [U'] \end{array}}{ext[T,T'](e_h \texttt{:} e_t) \ \vdash \ d_h \texttt{:} d_t}$$

This is true for stream constructions as well:

$$\frac{T = [U], T' = \text{Stream}(U')}{ext[T,T'](\texttt{[ ]}) \ \vdash \ \texttt{[ ]}_\texttt{S}}$$

$$\frac{\begin{array}{c} ext[U,U'](e_h) \ \vdash \ d_h, \ ext[T,\text{Task}(T')](e_t) \ \vdash \ d_t \\ T = [U], T' = \text{Stream}(U') \end{array}}{ext[T,T'](e_h \texttt{:} e_t) \ \vdash \ d_h \texttt{:}_\texttt{S} \ d_t}$$

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:

$$\frac{\begin{array}{c} ext[U_i, U'_i](e_i) \ \vdash \ d_i \\ T = (U_i), T' = (U'_i) \end{array}}{ext[T,T']((e_i)) \ \vdash (d_i)}$$

$$\frac{\begin{array}{c} ext[(U_1,\ldots,T,\ldots,U_n),(U_1,\ldots,T',\ldots,T_n)](e) \ \vdash \ d \\ \text{type}(e) = (U_1,\ldots,T,\ldots,U_n) \end{array}}{ext[T,T'](e\texttt{.}i) \ \vdash \ d\texttt{.}i}$$

In conditionals, both branches have to be extended. In definition blocks, only the block expression is affected.

$$\frac{ext[T,T'](e_t) \ \vdash \ d_t, \ ext[T,T'](e_f) \ \vdash \ d_f}{ext[T,T'](\texttt{if } e_c \texttt{ then } e_t \texttt{ else } e_f) \ \vdash \ \texttt{if } e_c \texttt{ then } d_t \texttt{ else } d_f}$$

$$\frac{ext[T,T'](e) \ \vdash \ d}{ext[T,T'](\texttt{let } \mathit{defs} \texttt{ in } e) \ \vdash \ \texttt{let } \mathit{defs} \texttt{ in } d}$$

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 guarantueed, 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 15 (Correctness)** *The result extension algorithm applied to an expression $e$ does not change the result of $e$.*

**Lemma 16 (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. □

## 4.11 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.

### 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 occurences of common synchronization expressions as follows:

$$\mathbf{CS}\left[\!\!\left[\begin{array}{l}\dots\\ \boxed{sync(e)\ \dots\ sync(e)}\\ \dots\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\dots\\ \texttt{let x = } sync(\mathrm{e})\\ \texttt{in } \boxed{x\ \dots\ x}\\ \dots\end{array}\right]\!\!\right]$$

The framed box here denotes the smallest expression that contains all occurences of the common synchronization operation (not considering occurences 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)
```

### Late Synchronizations

While the transformation algorithm defers scynchronization operations as much as possible, it does actually *not* guarantuee that a synchronization occurs at the latest possible moment. E.g. in an expression

$$f(\texttt{wait}(e_1),e_2)$$

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:

$$\mathbf{LS}\left[\!\!\left[\begin{array}{l}\dots\\ \boxed{sync(e_1)\ \dots\ f(e_2)}\\ \dots\end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l}\dots\\ \quad\texttt{let } x \texttt{ = } f(e_2)\\ \quad\texttt{in } \boxed{sync(e_1)\ \dots\ x}\\ \dots\end{array}\right]\!\!\right]$$

The frame denotes the largest expression that does neither contain a conditional nor a block expression surrounding both $sync(e_1)$ 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

$$\begin{array}{l} f\texttt{(...) =} \\ \quad \texttt{...} \\ \quad \boxed{e_2 \texttt{ : } f(e_1)} \end{array}$$

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.

### 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

$$e_1 \texttt{ + start(}f\texttt{, } e_2\texttt{)}$$

the `start` call should occur *before* $e_1$ is evaluated. The corresponding transformation is

$$\mathbf{ET}\left[\!\!\left[\begin{array}{l} \texttt{...} \\ \boxed{e_1 \texttt{ ... start(}f\texttt{, } e_2\texttt{)}} \\ \texttt{...} \end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l} \texttt{...} \\ \texttt{let } x \texttt{ = start(}f\texttt{, } e_2\texttt{)} \\ \texttt{in } \boxed{e_1 \texttt{...} x} \\ \texttt{...} \end{array}\right]\!\!\right]$$

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.

### 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

| | | | |
|---|---|---|---|
| *exp* | $\rightarrow$ | *composer iterator* | Composition |
| *composer* | $\rightarrow$ | `comp` $(f,\ b)$ | Composer |
| | $\mid$ | `list` | |
| | $\mid$ | `stream` | |
| | $\mid$ | `tcomp` $(f\ ,\ b,\ p)$ | |
| *iterator* | $\rightarrow$ | `[` $exp$ `|` $iter_1$ `|@`$par_1$ ... `|@`$par_n$ $iter_n$ `]` | Iterator List |

Figure 9: Parallel Compositions

```
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

$$\mathbf{OC}\left[\!\!\left[\begin{array}{l} \ldots \\ t\ \texttt{=}\ \texttt{start}(f,\ a_i) \\ \boxed{\texttt{wait}(t)} \\ \ldots \end{array}\right]\!\!\right] = \left[\!\!\left[\begin{array}{l} \ldots \\ \boxed{f(a_i)} \\ \ldots \end{array}\right]\!\!\right]$$

assuming that $t$ is not in use anywhere else.

# 5 Parallel Compositions

The non-deterministic composition construct described in Section 3 is only one of a set of parallel compositions. Several other forms are listed in Figure 9:

For each of these compositions, the programmer may indicate parallelism by iterator annotations `|@`$par_i$ where `|` is a shortcut for `|@(False)` and `|@` is a shortcut for `|@(True)`. The interpretation of such an annotation depends on the form of the composition (see also Figure 10):

1. **General:** In `comp` compositions, a task is started to generate and process the further bindings of the variables in $iter_i$. The current task recursively processes within the current bindings of $iter_i$ all remaining iterators $iter_{i+1}$ to $iter_n$ and so constructs a sequence of values. If necessary, it then synchronizes with the result of the child task and unwinds the recursion combining the task result with the generated values.

   Consequently, the individual bindings for the variables $iter_i$ are processed in parallel but the combinations are performed sequentially.

2. **Lists:** `list` compositions basically behave like `comp` compositions. However, due to the special nature of the `:` operator, each composition operation is performed immediately when a new value of the sequence is generated. The task then waits for the result list of the child task which is written into the tail of the last cell.

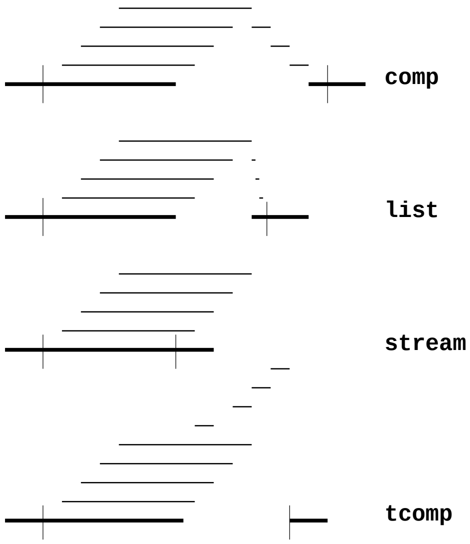


Figure 10: Parallel Composition Forms

Consequently, the individual list segments are truly generated in parallel and there is only a constant-time operation required to perform the combination.

3. **Streams:** A `stream` composition behave like `list` compositions but use `:s` onstructions to generate a stream of values. Consequently, the current task does not wait for the result of the child task but writes the task handle itself into the tail of the cell and immediately returns.

   Since streams are homogeneous data structures (the tail of every stream cell contains a task), every iterator basically requires an annotation `|@`. However, the programmer may skip the annotations since the transformation algorithm described in Section 4 will perform the necessary task extensions automatically.

4. **Tree Compositions:** In a composition `tcomp`, the combination function $f : T \times T \rightarrow T$ must be an associative operation. Similar to non-deterministic composition, the `|@`*par* annotation is used to split 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

   $$f((f(A_0, A_1), f(A_2, A_3)), \ldots, f(f(A_{n-4}, A_{n-3}), f(A_{n-2}, A_{n-1})))$$

   The optional termination predicate $p$ has the same role as in in non-deterministic compositions: it asserts that the result does not change any more and combination may be terminated prematurely.

In Figure 9, the dynamic behavior is illustrated assuming that the value sequence is partitioned into five clusters. The vertical lines indicate how long the main task (which processes the first cluster itself) is involved in the construct i.e. when execution starts and when the result is returned. In the `stream` case, the result contain other tasks that need not have terminated yet (this is in general the case for any `comp` operation with a combination function $f$ that is non-strict in its second argument).

Corresponding parallel versions of reverse composition operators (`rcomp`, `rlist`, `rstream`) are not yet supported, basically because their usefulness is questionable. A reverse variant `trcomp` of tree composition however will be provided in the future.

## 5.1 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
      snil => 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.

In the program

```
tcomp(prod, 1) [ y | x <- l, i <- 1.. |@(i%5==0) y <- fact(x) ]
```

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.

## 5.2 Translation of Parallel Compositions

In this sub section we define the general translation of a parallel composition `comb(`$f$`, `$b$`)`. The compositions `list` respectively `stream` are then defined as

```
stream = comp(:_S, [ ]_S)
list = comp(:, [ ])
```

Compared to the standard composition (see [Sch93a]), the translation of the parallel composition requires only a small modification depicted in Figure 11: we split the recursive branch by a conditional on *par* and annotate the recursive application of $F$ in that branch where *par* yields true.

Since the recursive application translates in this scheme into the argument of another function (finally into an argument of $f$), the transformation algorithm attempts to generate a new variant of this function where the task argument is not synchronized before the function is called.

If the current iterator is not the last one, this other function is the result of the translation of another iterator. In this case, the task argument needs only be synchronized in the base case and task evaluation can proceed in parallel with the recursion. In the case of the last iterator, however, the amount of parallelism depends on the nature of $f$. In the worst case, only the evaluation of $e$ proceeds in parallel with the child task.

### 5.3 Translation of Tree Compositions

The core translation of tree compositions is (almost) the same as that of non-deterministic compositions (see Section 3). The only difference is in the way that the individual subresults are combined by the auxiliary function `red` (see Figure 12):

1. `red` is called with the list of tasks generated by the composition construct; however contrary to the translation of `ndcomp` in Section 3 this list must be *reversed* before (to yield the original order of values).
2. `red` traverses the task list in a sequence of combination phases. During each phase, pairs of tasks are passed to a new task $F$ that synchronizes and combines their return values. Consequently, the main task creates (without any 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 effiently executed.

A reverse tree composition variant `trcomp` can be implemented by reversing the composition order in the core translation (by switching the bounds of arithmetic operators, reverting the lists for list iterators, and *not* inverting the binding lists of general iterators), and *not* reversing the task list to be passed to `red`.

## 6 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 third example demonstrates a real computer algebra application, namely the exact solution of a linear equation system over arbitrary precision integers.

The parallel programs were benchmarked on a Sequent Symmetry S81 computer with 20 processors i386 running at 20 MHz and equipped with 128 MB of main memory. Appendix A displays for every program the parallel target code generated by the compiler.

$$\mathbf{TC}[\![\texttt{comp}(f,b)\ \texttt{[}\ e\ \texttt{|}\ I\ \texttt{]}]\!] = \mathbf{TI}[\![I]\!][\![f]\!][\![e]\!][\![b]\!]$$

$$\mathbf{TI}[\![\ ]\!][\![f]\!][\![e]\!][\![v]\!] \ = \ [\![f\,(e,v)]\!]$$

$$\mathbf{TI}[\![h_i\texttt{:}t_i\texttt{<-}l_i\texttt{,}\ x_j\texttt{<-}a_j\texttt{..}b_j\texttt{+=}c_j\texttt{,}\ y_k\texttt{=}r_k\texttt{,}\ defs_1\texttt{;}\ s\texttt{;}\ y_k\texttt{'} = t_k\texttt{,}\ defs_2\ \texttt{|@}par\ R]\!]$$
$$[\![f]\!][\![e]\!][\![v]\!] \ =$$

```
[[ let
     F(z1, ..., zm, x1, ..., xn, y1, ..., yo) =
       case z1 of
         [ ] => v̄
         h1:t1 =>
           ...
             case zm of
               [ ] => v̄
               hm:tm =>
                 if x1 > b̄1 || ... || xn > b̄n
                   then v̄
                   else let
                     defs1
                   in if ^s
                     then v̄
                     else let
                       defs2
                     in if par then
                       TI[[R]][[f]][[e]]
                       [[F(t1, ..., tm, x1+c̄1, ..., xn+c̄n, r1, ..., ro)@]]
                     else
                       TI[[R]][[f]][[e]]
                       [[F(t1, ..., tm, x1+c̄1, ..., xn+c̄n, r1, ..., ro)]]
     b̄j =bj; c̄j = cj; v̄ = v
   in
     F(l1, ..., lm, a1, ..., an, u1, ..., uo)                               ]]
```

Figure 11: Parallel Composition

```
red(l) =
  case l of
    [ ] => (False, v)
    h1:t1 =>
      case t1 of
        [ ] => wait(h1)
        h2:t2 => red(start(F2, h1, h2) : red0(t2))

red0(l) =
  case l of
    [ ] => [ ]
    h1:t1 => case t1 of
      [ ] => [h1]
      h2:t2 => start(F2, h1, h2) : red0(t2)

F(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 12: Tree Composition (Auxiliaries)

```
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 13: Prime Numbers (1. 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 14: Prime Numbers (2. Variant)

## 6.1 Prime Numbers

Figure 13 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 14. 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.

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 progrma 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 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 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 interpretes 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 15 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 [Cro94] by giving the absolute execution times and using logarithmic scales for both axes of the diagram. The dotted lines illustrate linear speedups.

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 practically as fast as a corresponding SACLIB C program.

## 6.2 N Queens

Figure 16 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 (superflously) 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!@
```

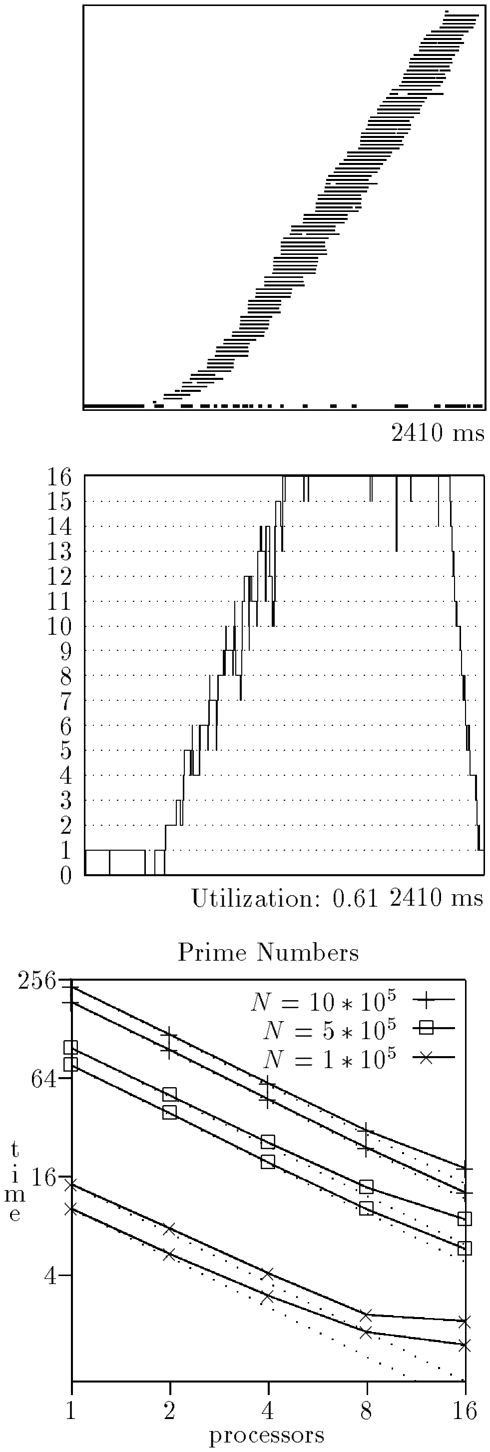


Figure 15: Prime Numbers

```
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 16: N Queens

```
        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

```
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 17 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 very similar except with smaller $N$ and a large number of processors. In this case, the second variant (sequential concatenation) 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.

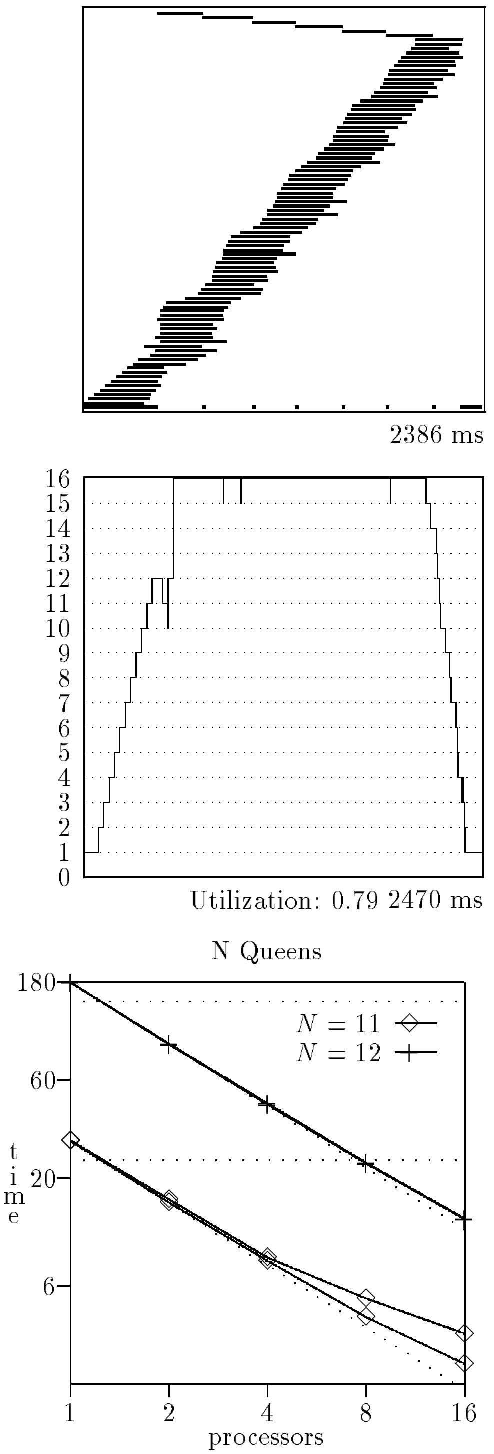


Figure 17: N Queens

```
-----------------------------------------------------------------------------
--
-- 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)

}
```

Figure 18: 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
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)
```

Figure 19: Linear Equations (Part 2)

### 6.3 Linear Equations

Figures 18 and 19 demonstrate a parallel variant of the modular algorithm for the solution of a linear equation system; the sequential version was presented in [Sch93a]:

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 [Sta92].

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 are used to construct the modular domains.
2. The main task starts a set of `linDets` tasks. Each task $k$ projects the equation system $(A, b)$ into the modular domain determined by $p_k$ and computes $(d_k, \langle y_{k1} \ldots y_{kn} \rangle)$. The prime $p_k$, 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 `crad` (returning a value $d$), a task that starts $n$ subtasks `cray` (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 is not the case, the very unlikely case of an "unlucky prime" $p_i$ has occured. 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.

Figure 20 illustrates the runtime behavior of the parallel program for several values of $D$ and $L$ 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 `cray/cra` (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. These results are essentially the same as for the manually coded PACLIB parallelization [Loi93].

## 7 Conclusions and Further Work

The para-functional programming interface presented in this paper allows the programmer to write parallel programs in several levels of abstractions that are compatible with each other:

1. On the imperative level in the PACLIB C language. This gives the most flexibility but also requires much low-level coding effort. 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 (i.e. no runtime checks are inserted).

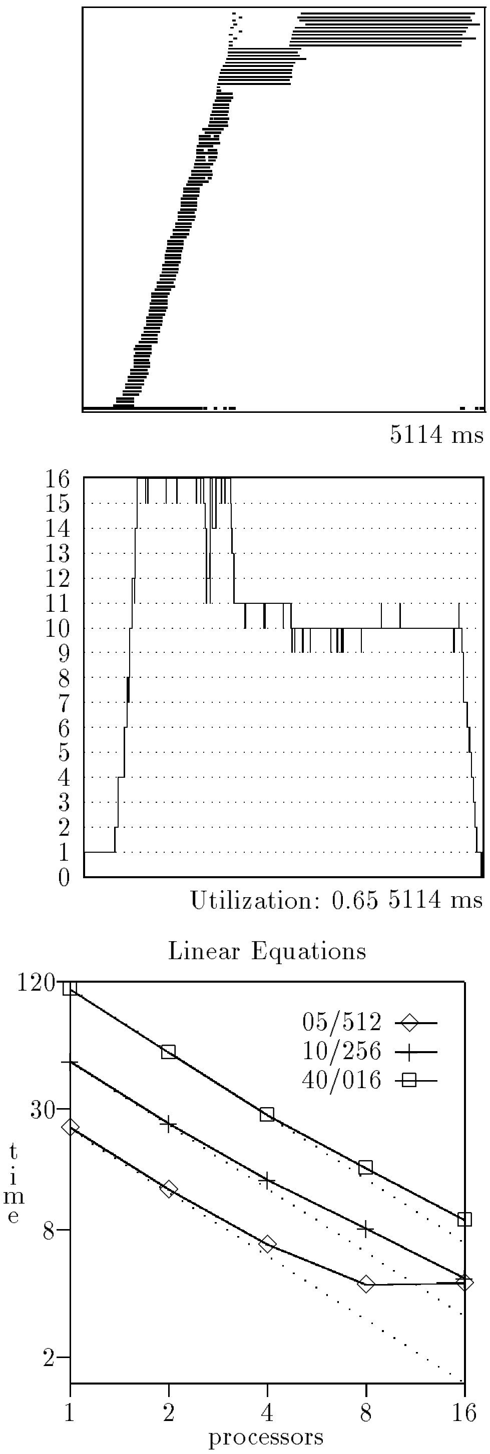


Figure 20: Linear Equations

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 parallelization (or just para-type) annotations without explicitly creating `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).

The main problem with this approach is that there is no directly visible relationship between the denotational semantics of a program and its parallel operational behavior. However, we believe that once the programmer has understood the basic concepts, the system is fairly easy to use for writing efficient parallel programs.

In particular it is important to understand what the compiler does (caring for the details of task creation and synchronization) and what it does not (transforming a sequential program into a parallel one). If the programmer cares for the second and gives some focused hints, the compiler will care for the first in a rather intelligent fashion.

Our further work will concentrate on the minimization of these hints. The idea is to give the programmer the possibility to annotate functions with the (asymptotical) complexity of their execution time and result size. The compiler will attempt (by analysis in combination with profiling) to determine the actual measures and apply this information to create parallelization annotations at promising places (in particular at iterated compositions). This would again shift the information provided by the programmer to a higher level of abstraction.

Another area for further extension is to retarget the compiler from PACLIB to the multiprocessor kernel of the newly developed STURM library [HNSS94]. Since this successor of PACLIB is builit on top of arrays instead of lists this requires the introduction of array processing into the language.

## References


[Ble93] 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.

[Che85] Marina C. Chen. Crystal: A Synthesis Approach to Programming Parallel Machines. In Michael T. Heath, editor, *Hypercube Multiprocessors 1986; Proceedings of the First Conference on Hypercube Multiprocessors*, pages 87–107, Knoxville, Tennessee, August 26–27, 1985, 1985. SIAM, Philadelphia.

[Col89] Murray I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, London, 1989.

[Cro94] Lawrence A. Crowl. How to Measure, Present, and Compare Parallel Performance. *IEEE Parallel and Distributed Technology*, 2(1):9–25, Spring 1994.

[DFH+93] 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 *PARLE 93: Parallel Languages and Architectures Europe*, Munich, Germany, June 14–18, 1993.

[DOPV94] 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.

[FCO90] Feo, Cann, and Oldehoeft. A Report on the Sisal Language Project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, December 1990.

[GG89] Ron Goldman and Richard P. Gabriel. QLisp: Parallel Processing in Lisp. *IEEE Software*, pages 51–59, July 1989.

[Hal85] Robert H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[Hil93] Jonathan M. D. Hill. Data Parallel Haskell: Mixing Old and New Glue. Technical report, Queen Mary and Westfield College, University of London, UK, 1993.

[HNS93] 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.

[HNSS94] Hoon Hong, Andreas Neubacher, Wolfgang Schreiner, and Volker Stahl. The C++ Interface to the STURM Multi-Processor Kernel. Technical Report 94-32, RISC-Linz, Johannes Kepler University, Linz, Austria, April 1994.

[Hud86] Paul Hudak. Para-Functional Programming. *IEEE Computer*, 19(8):60–70, August 1986.

[Hud91] 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.

[Kel89] Paul Kelly. *Functional Programming for Loosely-coupled Multiprocessors*. Research Monographs in Parallel and Distributed Computing. Pitman, London, 1989.

[KG89] 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.

[Loi93] 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.

[Mer91] Simon Merrall. TPL — An Implementation of Paralation Lisp in EuLisp. Technical report, School of Mathematical Sciences, University of Bath, 1991.

[NAH93] 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.

[Nik91] R. S. Nikhil. Id (Version 90.1) Reference Manual. CSG Memo 284-2, MIT Laboratory for Computer Science, Cambridge, MA, July 1991.

[NSEP91] 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.

[Rab93] Fethi A. Rabhi. Functional Languages and Parallelism. In *3rd Workshop on Parallel Computing*, Polytechnique de Mons, Belgium, September 23–24, 1993.

[Sch93a] Wolfgang Schreiner. Compiling a Functional Language to Efficient SACLIB C. Technical Report 93-49, RISC-Linz, Johannes Kepler University, Linz, Austria, September 1993.

[Sch93b] Wolfgang Schreiner. Parallel Functional Programming — An Annotated Bibliography. Technical Report 93-24, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1993.

[Sch94] Wolfgang Schreiner. Virtual Tasks for the PACLIB Kernel. In *CONPAR 94 - VAPP VI International Conference on Parallel and Vector Processing*, Linz, Austria, September 6–8, 1994. Springer, Berlin.

[SJH86] Guy L. Steele Jr. and W. Daniel Hillis. Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 279–297, Cambridge, Massachusetts, August 4–6, 1986. ACM Press.

[Sta92] Volker Stahl. Solving a System of Linear Equations with Modular Arithmetic on a MIMD Computer. Technical Report 92-62, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992. Also: Technical Report ACPC/TR 93-5, ACPC Technical Report Series, Austrian Center for Parallel Computation, February 1993.

[WA85] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language*, volume 22 of *A.P.I.C. Studies in Data Processing*. Academic Press, London, 1985.

[Wad91] 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.

[ZHHL87] Benjamin Zorn, Paul Hilfinger, Kinson Ho, and James Larus. SPUR Lisp: Design and Implementation. Technical Report UCB/CSD 87/373, Computer Science Division (EECS), University of California, Berkeley, CA, September 1987.

# A Generated Target Code

In this section, we present the compiler-generated target code of the program examples presented in Section 6. Only the executable code is given; in particular the type declarations are stripped off.

## A.1 Prime Numbers (Basic Circular Variant)

```
/*
 * local function definitions
 */

static inline STREAM_54 test_270(INT _271, TSTREAM_49 _272, TSTREAM_49 _273)
{
  while (TRUE) {
    STREAM_54 _454 = WAIT(_272);
    if (ISNIL(_454))
      ERROR("test");
    else {
      TSTREAM_49 pl_426 = TAIL(_454);
      INT p_425 = HEAD(_454);
      if (_271 < (p_425 * p_425))
        return (CONS(_271, _273));
      else if ((_271 % p_425) == 0) {
        STREAM_54 _453 = WAIT(_273);
        return (_453);
      } else
        _272 = pl_426;
    }
  }
}

static inline STREAM_54 fun_210(INT _211, INT _212, TSTREAM_49 _213)
{
  INT _320 = (_212 + 2);
  if (_320 >= _211)
    return (NIL);
  else {
    TSTREAM_49 _455 = START3(fun_210, _211, _320, _213);
    return (test_270(_320, _213, _455));
  }
}

static inline STREAM_54 fun_111(INT N_112, TSTREAM_49 plist_113)
{
  if (5 >= N_112)
    return (NIL);
  else {
    TSTREAM_49 _456 = START3(fun_210, N_112, 5, plist_113);
    return (test_270(5, plist_113, _456));
  }
}

static inline STREAM_54 fun_139(void)
{
  return (NIL);
}

static inline STREAM_54 fun_126(INT N_127, TSTREAM_49 plist_128)
{
  TSTREAM_49 _457 = START2(fun_111, N_127, plist_128);
  return (CONS(3, _457));
}

static inline void streamsync_465(STREAM_54 _93, LIST_45 * _466)
{
```

```
  while (!ISNIL(_93)) {
    STREAM_54 _458 = WAIT(TAIL(_93));
    _466 = RCONS(HEAD(_93), _466);
    _93 = _458;
  }
  *_466 = NIL;
  return;
}

static inline LIST_45 streamsync_92(STREAM_54 _93)
{
  LIST_45 _466;
  streamsync_465(_93, &_466);
  return (_466);
}

static inline STREAM_54 fun_70(LIST_45 _71)
{
  LIST_45 _391 = TAIL(_71);
  if (ISNIL(_391))
    return (NIL);
  else {
    TSTREAM_49 _459 = START1(fun_70, _391);
    return (CONS(HEAD(_391), _459));
  }
}

/*
 * external entry function
 */

LIST_45 dcmain(INT N)
{
  if (N <= 3) {
    STREAM_54 _448 = fun_139();
    return (streamsync_92(_448));
  } else {
    TSTREAM_49 plist_446 = FORWARD();
    TSTREAM_49 _451 = FSTART2(plist_446, fun_126, N, plist_446);
    STREAM_54 _450 = WAIT(plist_446);
    return (streamsync_92(_450));
  }
}

/*
 * End of file
 */
```

## A.2 Prime Numbers (Cyclic Cluster Variant)

```
/*
 * local function definitions
 */

static inline IntList_34 test_667(IntList_34 _668, TSTREAM_427 _669, INT _670,
                                  IntList_34 _671)
{
  while (TRUE)
    if (ISNIL(_668)) {
      STREAM_487 _808 = WAIT(_669);
      if (ISNIL(_808))
        ERROR("test");
      else {
        _669 = TAIL(_808);
        _668 = HEAD(_808);
      }
    } else {
```

```
      INT p_782 = HEAD(_668);
      if (_670 < (p_782 * p_782))
        return (CONS(_670, _671));
      else if ((_670 % p_782) == 0)
        return (_671);
      else
        _668 = TAIL(_668);
    }
}

static inline IntList_34 loop_588(INT _589, INT _590, TSTREAM_427 _591,
                                  INT _592)
{
  if (_589 <= (_590 + _592))
    return (test_667(NIL, _591, _589, loop_588((_589 + 1), _590, _591, _592)));
  else
    return (NIL);
}

static inline STREAM_487 fun_679(INT _680, INT _681, TSTREAM_427 _682,
                                 INT _683)
{
  INT _719 = (_681 + _683);
  if (_719 >= _680)
    return (NIL);
  else {
    TSTREAM_427 _809 = START4(fun_679, _680, _719, _682, _683);
    return (CONS(loop_588(_719, _719, _682, _683), _809));
  }
}

static inline STREAM_487 fun_577(void)
{
  return (NIL);
}

static inline STREAM_487 fun_561(INT N_562, INT M_563, TSTREAM_427 plist_564)
{
  TSTREAM_427 _810 = START3(fun_552, N_562, M_563, plist_564);
  return (CONS(base(M_563), _810));
}

static inline STREAM_487 fun_552(INT N_553, INT M_554, TSTREAM_427 plist_555)
{
  if (M_554 >= N_553)
    return (NIL);
  else {
    TSTREAM_427 _811 = START4(fun_679, N_553, M_554, plist_555, M_554);
    return (CONS(loop_588(M_554, M_554, plist_555, M_554), _811));
  }
}

static inline void loop_819(IntList_34 xl_452, IntList_34 l2_476,
                            IntList_34 * _820)
{
  while (!ISNIL(xl_452)) {
    _820 = RCONS(HEAD(xl_452), _820);
    xl_452 = TAIL(xl_452);
  }
  *_820 = l2_476;
  return;
}

static inline IntList_34 loop_455(IntList_34 xl_452, IntList_34 l2_476)
{
  IntList_34 _820;
  loop_819(xl_452, l2_476, &_820);
  return (_820);
}
```

```
static inline IntList_34 loop_579(TSTREAM_427 _580)
{
  STREAM_487 _812 = WAIT(_580);
  if (ISNIL(_812))
    return (NIL);
  else
    return (loop_455(HEAD(_812), loop_579(TAIL(_812))));
}

/*
 * external entry function
 */

IntList_34 dcmain(INT N, INT M)
{
  if (N < 3) {
    TSTREAM_427 _806 = START0(fun_577);
    return (loop_579(_806));
  } else {
    TSTREAM_427 plist_804 = FORWARD();
    TSTREAM_427 _807 = FSTART3(plist_804, fun_561, N, M, plist_804);
    return (loop_579(plist_804));
  }
}

/*
 * End of file
 */
```

## A.3 Primes (Non-Cyclic Cluster Variant)

```
/*
 * local function definitions
 */

static inline LIST_60 test_285(LIST_60 _286, TSTREAM_71 _287, INT _288,
  LIST_60 _289)
{
  while (TRUE)
    if (ISNIL(_286)) {
      STREAM_107 _422 = WAIT(_287);
      if (ISNIL(_422))
        ERROR("test");
      else {
        _287 = TAIL(_422);
        _286 = HEAD(_422);
      }
    } else {
      INT p_391 = HEAD(_286);
      if (_288 < (p_391 * p_391))
        return (CONS(_288, _289));
      else if ((_288 % p_391) == 0)
        return (_289);
      else
        _286 = TAIL(_286);
    }
}

static inline LIST_60 loop_203(INT _204, INT _205, TSTREAM_71 _206)
{
  if (_204 <= (_205 + 1000))
    return (test_285(NIL, _206, _204, loop_203((_204 + 1), _205, _206)));
  else
    return (NIL);
}
```

```
static inline STREAM_107 fun_297(INT _298, INT _299, TSTREAM_71 _300)
{
  INT _334 = (_299 + 1000);
  if (_334 >= _298)
    return (NIL);
  else {
    TSTREAM_71 _423 = START3(fun_297, _298, _334, _300);
    return (CONS(loop_203(_334, _334, _300), _423));
  }
}

static inline void append_430(LIST_60 _194, TSTREAM_71 _195, LIST_60 * _431)
{
  while (TRUE)
    if (ISNIL(_194)) {
      STREAM_107 _424 = WAIT(_195);
      if (ISNIL(_424)) {
        *_431 = NIL;
        return;
      } else {
        _195 = TAIL(_424);
        _194 = HEAD(_424);
      }
    } else {
      _431 = RCONS(HEAD(_194), _431);
      _194 = TAIL(_194);
    }
}

static inline LIST_60 append_193(LIST_60 _194, TSTREAM_71 _195)
{
  LIST_60 _431;
  append_430(_194, _195, &_431);
  return (_431);
}

static inline STREAM_107 fun_191(void)
{
  return (NIL);
}

static inline STREAM_107 fun_174(INT N_175, LIST_60 B_176, TSTREAM_71
  plist_177)
{
  TSTREAM_71 _425 = START2(fun_165, N_175, plist_177);
  return (CONS(B_176, _425));
}

static inline STREAM_107 fun_165(INT N_166, TSTREAM_71 plist_167)
{
  if (1000 >= N_166)
    return (NIL);
  else {
    TSTREAM_71 _426 = START3(fun_297, N_166, 1000, plist_167);
    return (CONS(loop_203(1000, 1000, plist_167), _426));
  }
}

/*
 * external entry function
 */

LIST_60 dcmain(INT N)
{
  if (N < 3) {
    TSTREAM_71 _420 = START0(fun_191);
    return (append_193(NIL, _420));
  } else {
    TSTREAM_71 plist_418 = FORWARD();
```

```
    TSTREAM_71 _421 = FSTART3(plist_418, fun_174, N, base(1000), plist_418);
    return (append_193(NIL, plist_418));
  }
}

/*
 * End of file
 */
```

## A.4 N Queens (Basic Variant)

```
/*
 * local function definitions
 */

static inline BOOL conflict_27(INT col_40, INT dist_41, Board_23 crest_42)
{
  while (!ISNIL(crest_42)) {
    INT col0_193 = HEAD(crest_42);
    if ((col_40 == col0_193) || ((col_40 == (col0_193 + dist_41)) ||
        (col_40 == (col0_193 - dist_41))))
      return (TRUE);
    else {
      crest_42 = TAIL(crest_42);
      dist_41 = (dist_41 + 1);
    }
  }
  return (FALSE);
}

static inline LIST_53 solve_81(INT _82, INT _83, INT _84, Board_23 _85,
                               TASK_70 _86)
{
  while (!(_83 > _82)) {
    TASK_70 _245 = START5(fun_125, _82, _83, _84, _85, _86);
    if (conflict_27(_83, 1, _85)) {
      LIST_53 _243 = WAIT(_245);
      return (_243);
    } else if (_84 == _82) {
      LIST_53 _244 = WAIT(_245);
      return (CONS(CONS(_83, _85), _244));
    } else {
      _86 = _245;
      _85 = CONS(_83, _85);
      _84 = (_84 + 1);
      _83 = 1;
    }
  }
  {
    LIST_53 _242 = WAIT(_86);
    return (_242);
  }
}

static inline LIST_53 fun_125(INT _126, INT _127, INT _128, Board_23 _129,
                              TASK_70 _130)
{
  return (solve_81(_126, (_127 + 1), _128, _129, _130));
}

static inline LIST_53 fun_64(INT n_65, INT col_66, INT row_67,
                             Board_23 part_68, LIST_53 rest_69)
{
  INT col_153 = (col_66 + 1);
  if (col_153 > n_65)
    return (rest_69);
```

```
  else {
    TASK_70 _248 = START5(fun_64, n_65, col_153, row_67, part_68, rest_69);
    if (conflict_27(col_153, 1, part_68)) {
      LIST_53 _246 = WAIT(_248);
      return (_246);
    } else if (row_67 == n_65) {
      LIST_53 _247 = WAIT(_248);
      return (CONS(CONS(col_153, part_68), _247));
    } else
      return (solve_81(n_65, 1, (row_67 + 1), CONS(col_153, part_68), _248));
  }
}

/*
 * external entry function
 */

LIST_53 dcmain(INT N)
{
  if (1 > N)
    return (NIL);
  else {
    TASK_70 _241 = START5(fun_64, N, 1, 1, NIL, NIL);
    if (conflict_27(1, 1, NIL)) {
      LIST_53 _239 = WAIT(_241);
      return (_239);
    } else if (1 == N) {
      LIST_53 _240 = WAIT(_241);
      return (CONS(CONS(1, NIL), _240));
    } else
      return (solve_81(N, 1, 2, CONS(1, NIL), _241));
  }
}

/*
 * End of file
 */
```

## A.5 N Queens (Tree Concatenation)

```
/*
 * local function prototypes
 */

static inline BOOL conflict_27(INT, INT, Board_23);
static inline LIST_54 fun_146(INT, INT, INT, Board_23, TASK_85, INT);
static inline LIST_54 solve_96(INT, INT, INT, Board_23, TASK_85, INT);
static inline LIST_54 solve_25(INT, INT, INT, Board_23, LIST_54, INT);
static inline LIST_54 fun_78(INT, INT, INT, Board_23, LIST_54, INT);

/*
 * local function definitions
 */

static inline BOOL conflict_27(INT col_41, INT dist_42, Board_23 crest_43)
{
  while (!ISNIL(crest_43)) {
    INT col0_192 = HEAD(crest_43);
    if ((col_41 == col0_192) || ((col_41 == (col0_192 + dist_42)) ||
        (col_41 == (col0_192 - dist_42))))
      return (TRUE);
    else {
      crest_43 = TAIL(crest_43);
      dist_42 = (dist_42 + 1);
    }
  }
  return (FALSE);
```

```
}

static inline LIST_54 fun_146(INT _147, INT _148, INT _149, Board_23 _150,
                              TASK_85 _151, INT _152)
{
  return (solve_96(_147, (_148 + 1), _149, _150, _151, _152));
}

static inline void solve_312(INT _97, INT _98, INT _99, Board_23 _100,
                             TASK_85 _101, INT _102, LIST_54 * _313)
{
  while (!(_98 > _97))
    if (_99 < _102) {
      TASK_85 _306 = START6(fun_146, _97, _98, _99, _100, _101, _102);
      if (conflict_27(_98, 1, _100)) {
        LIST_54 _304 = WAIT(_306);
        *_313 = _304;
        return;
      } else if (_99 == _97) {
        LIST_54 _305 = WAIT(_306);
        *_313 = CONS(CONS(_98, _100), _305);
        return;
      } else {
        _101 = _306;
        _100 = CONS(_98, _100);
        _99 = (_99 + 1);
        _98 = 1;
      }
    } else if (conflict_27(_98, 1, _100))
      _98 = (_98 + 1);
    else if (_99 == _97) {
      _313 = RCONS(CONS(_98, _100), _313);
      _98 = (_98 + 1);
    } else {
      *_313 = solve_25(_97, 1, (_99 + 1), CONS(_98, _100),
                       solve_96(_97, (_98 + 1), _99, _100, _101, _102), _102);
      return;
    }
  {
    LIST_54 _303 = WAIT(_101);
    *_313 = _303;
    return;
  }
}

static inline LIST_54 solve_96(INT _97, INT _98, INT _99, Board_23 _100,
                               TASK_85 _101, INT _102)
{
  LIST_54 _313;
  solve_312(_97, _98, _99, _100, _101, _102, &_313);
  return (_313);
}

static inline void solve_314(INT n_29, INT col_30, INT row_31,
  Board_23 part_32, LIST_54 rest_33, INT P_60, LIST_54 * _315)
{
  while (!(col_30 > n_29))
    if (row_31 < P_60) {
      TASK_85 _309 = START6(fun_78, n_29, col_30, row_31, part_32, rest_33, P_60);
      if (conflict_27(col_30, 1, part_32)) {
        LIST_54 _307 = WAIT(_309);
        *_315 = _307;
        return;
      } else if (row_31 == n_29) {
        LIST_54 _308 = WAIT(_309);
        *_315 = CONS(CONS(col_30, part_32), _308);
        return;
      } else {
        *_315 = solve_96(n_29, 1, (row_31 + 1), CONS(col_30, part_32), _309,
```

```
                         P_60);
        return;
      }
    } else if (conflict_27(col_30, 1, part_32))
      col_30 = (col_30 + 1);
    else if (row_31 == n_29) {
      _315 = RCONS(CONS(col_30, part_32), _315);
      col_30 = (col_30 + 1);
    } else {
      rest_33 = solve_25(n_29, (col_30 + 1), row_31, part_32, rest_33, P_60);
      part_32 = CONS(col_30, part_32);
      row_31 = (row_31 + 1);
      col_30 = 1;
    }
  *_315 = rest_33;
  return;
}

static inline LIST_54 solve_25(INT n_29, INT col_30, INT row_31,
  Board_23 part_32, LIST_54 rest_33, INT P_60)
{
  LIST_54 _315;
  solve_314(n_29, col_30, row_31, part_32, rest_33, P_60, &_315);
  return (_315);
}

static inline LIST_54 fun_78(INT n_79, INT col_80, INT row_81,
  Board_23 part_82, LIST_54 rest_83, INT P_84)
{
  return (solve_25(n_79, (col_80 + 1), row_81, part_82, rest_83, P_84));
}

/*
 * external entry function
 */

LIST_54 dcmain(INT N, INT P)
{
  return (solve_25(N, 1, 1, NIL, NIL, P));
}

/*
 * End of file
 */
```

## A.6 N Queens (Sequential Concatenation)

```
/*
 * local function definitions
 */

static inline void extend_509(LIST_65 _101, LIST_93 * _510)
{
  while (!ISNIL(_101)) {
    TASK_92 _503 = START1(fun_105, _101);
    _510 = RCONS(_503, _510);
    _101 = TAIL(_101);
  }
  *_510 = NIL;
  return;
}

static inline LIST_93 extend_100(LIST_65 _101)
{
  LIST_93 _510;
  extend_509(_101, &_510);
  return (_510);
```

```
}

static inline LIST_63 fun_108(INT col_109, Board_24 part_110)
{
  return (CONS(CONS(col_109, part_110), NIL));
}

static inline LIST_63 fun_105(LIST_65 _106)
{
  return (HEAD(_106));
}

static inline BOOL conflict_28(INT col_43, INT dist_44, Board_24 crest_45)
{
  while (!ISNIL(crest_45)) {
    INT col0_397 = HEAD(crest_45);
    if ((col_43 == col0_397) || ((col_43 == (col0_397 + dist_44)) || (col_43
        == (col0_397 - dist_44))))
      return (TRUE);
    else {
      crest_45 = TAIL(crest_45);
      dist_44 = (dist_44 + 1);
    }
  }
  return (FALSE);
}

static inline void solve_514(INT n_37, INT col_38, INT row_39, Board_24
  part_40, LIST_63 rest_41, LIST_63 * _515)
{
  while (!(col_38 > n_37))
    if (conflict_28(col_38, 1, part_40))
      col_38 = (col_38 + 1);
    else if (row_39 == n_37) {
      _515 = RCONS(CONS(col_38, part_40), _515);
      col_38 = (col_38 + 1);
    } else {
      rest_41 = solve_27(n_37, (col_38 + 1), row_39, part_40, rest_41);
      part_40 = CONS(col_38, part_40);
      row_39 = (row_39 + 1);
      col_38 = 1;
    }
  *_515 = rest_41;
  return;
}

static inline LIST_63 solve_27(INT n_37, INT col_38, INT row_39, Board_24
  part_40, LIST_63 rest_41)
{
  LIST_63 _515;
  solve_514(n_37, col_38, row_39, part_40, rest_41, &_515);
  return (_515);
}

static inline LIST_63 fun_86(INT n_87, INT col_88, INT row_89, Board_24
  part_90)
{
  return (solve_27(n_87, col_88, row_89, part_90, NIL));
}

static inline LIST_93 psolve_303(INT _304, INT _305, INT _306, Board_24 _307,
  LIST_93 _308)
{
  while (!(_306 > 2))
    if (_305 > _304)
      return (_308);
    else {
      LIST_93 sols_458 = psolve_303(_304, (_305 + 1), _306, _307, _308);
      if (conflict_28(_305, 1, _307))
```

```
        return (sols_458);
      else if (_306 == _304) {
        TASK_92 _505 = START2(fun_108, _305, _307);
        return (CONS(_505, sols_458));
      } else {
        _308 = sols_458;
        _307 = CONS(_305, _307);
        _306 = (_306 + 1);
        _305 = 1;
      }
    }
  {
    TASK_92 _504 = START4(fun_86, _304, _305, _306, _307);
    return (CONS(_504, _308));
  }
}

static inline LIST_93 psolve_137(INT n_138, INT col_139, INT row_140, Board_24
  part_141, LIST_65 rest_142)
{
  if (row_140 > 2) {
    TASK_92 _506 = START4(fun_86, n_138, col_139, row_140, part_141);
    return (CONS(_506, extend_100(rest_142)));
  } else if (col_139 > n_138)
    return (extend_100(rest_142));
  else {
    LIST_93 sols_491 = psolve_137(n_138, (col_139 + 1), row_140, part_141,
                                  rest_142);
    if (conflict_28(col_139, 1, part_141))
      return (sols_491);
    else if (row_140 == n_138) {
      TASK_92 _507 = START2(fun_108, col_139, part_141);
      return (CONS(_507, sols_491));
    } else
      return (psolve_303(n_138, 1, (row_140 + 1), CONS(col_139, part_141),
                         sols_491));
  }
}

static inline LIST_63 append_209(TASK_92 _210, LIST_93 _211)
{
  while (TRUE) {
    LIST_63 _508 = WAIT(_210);
    if (ISNIL(_508))
      if (ISNIL(_211))
        return (NIL);
      else {
        _210 = HEAD(_211);
        _211 = TAIL(_211);
    } else
      return (CONS(HEAD(_508), append_159(TAIL(_508), _211)));
  }
}

static inline void append_520(LIST_63 _160, LIST_93 _161, LIST_63 * _521)
{
  while (TRUE)
    if (ISNIL(_160))
      if (ISNIL(_161)) {
        *_521 = NIL;
        return;
      } else {
        *_521 = append_209(HEAD(_161), TAIL(_161));
        return;
    } else {
      _521 = RCONS(HEAD(_160), _521);
      _160 = TAIL(_160);
    }
}
```

```
static inline LIST_63 append_159(LIST_63 _160, LIST_93 _161)
{
  LIST_63 _521;
  append_520(_160, _161, &_521);
  return (_521);
}

/*
 * external entry function
 */

LIST_63 dcmain(INT N)
{
  return (append_159(NIL, psolve_137(N, 1, 1, NIL, NIL)));
}

/*
 * End of file
 */
```

## A.7 Linear Equation Solver

```
/*
 * local function definitions
 */

static inline Rational fun_1160(LIST_840 _1161, TASK_839 _1162)
{
  TASK_839 _1509 = HEAD(_1161);
  Integer _2088 = WAIT(_1509);
  Integer _2089 = WAIT(_1162);
  return (RNRED(_2088, _2089));
}

static inline BOOL loop_937(LIST_778 _938)
{
  while (!ISNIL(_938)) {
    DigitPair_114 _2090 = WAIT(HEAD(_938));
    if (ELEM1(_2090) != 0)
      _938 = TAIL(_938);
    else
      return (FALSE);
  }
  return (TRUE);
}

static inline Integer loop_893(IntList_44 _894, IntList_44 _895,
                               IntegerList_107 _896, LIST_778 _897,
                               Integer _898)
{
  while (!ISNIL(_894))
    if (ISNIL(_895))
      return (_898);
    else if (ISNIL(_896))
      return (_898);
    else if (ISNIL(_897))
      return (_898);
    else {
      LIST_778 at_1786 = TAIL(_897);
      DigitPair_114 _2091 = WAIT(HEAD(_897));
      INT d_1785 = ELEM1(_2091);
      Integer M_1783 = HEAD(_896);
      IntegerList_107 Mt_1782 = TAIL(_896);
      INT mi_1781 = HEAD(_895);
      IntList_44 mit_1780 = TAIL(_895);
```

```
      INT m_1779 = HEAD(_894);
      IntList_44 mt_1778 = TAIL(_894);
      if (d_1785 != 0) {
        _898 = MIDCRA(M_1783, m_1779, mi_1781, _898, d_1785);
        _897 = at_1786;
        _896 = Mt_1782;
        _895 = mit_1780;
        _894 = mt_1778;
      } else
        return (_898);
    }
  return (_898);
}

static inline Integer fun_904(LIST_778 dkyk_905, IntList_44 pk_906,
                              IntegerList_107 Pk_907, IntList_44 ik_908)
{
  return (loop_893(pk_906, ik_908, Pk_907, dkyk_905, DTOI(1)));
}

static inline void listsync_2106(LIST_740 _765, RatVector_111 * _2107)
{
  while (!ISNIL(_765)) {
    Rational _2092 = WAIT(HEAD(_765));
    _2107 = RCONS(_2092, _2107);
    _765 = TAIL(_765);
  }
  *_2107 = NIL;
  return;
}

static inline RatVector_111 listsync_764(LIST_740 _765)
{
  RatVector_111 _2107;
  listsync_2106(_765, &_2107);
  return (_2107);
}

static inline Integer loop_616(IntegerList_107 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_626(IntegerList_107 xl_623, Integer _625)
{
  while (!ISNIL(xl_623)) {
    _625 = IMAX(IABSF(HEAD(xl_623)), _625);
    xl_623 = TAIL(xl_623);
  }
  return (_625);
}

static inline Integer loop_622(IntMatrix_110 ll_619, Integer _621)
{
  while (!ISNIL(ll_619)) {
    _621 = loop_626(HEAD(ll_619), _621);
    ll_619 = TAIL(ll_619);
  }
  return (_621);
}

static inline void loop_2111(IntList_44 pl_797, Integer P_798,
  IntMatrix_110 A_799, IntegerList_107 b_800, INT n_801, Integer B_802,
  LIST_778 * _2112, IntList_44 * _2113, IntegerList_107 * _2114,
  IntList_44 * _2115, IntList_44 * _2116)
```

```
{
  if (ISNIL(pl_797)) {
    *_2112 = NIL;
    *_2113 = NIL;
    *_2114 = NIL;
    *_2115 = NIL;
    *_2116 = NIL;
    return;
  } else if (ICOMP(P_798, B_802) == -1) {
    LIST_778 _2117;
    IntList_44 _2118;
    IntegerList_107 _2119;
    IntList_44 _2120;
    IntList_44 _2121;
    INT p_1833 = HEAD(pl_797);
    TASK_777 _2093 = START4(linDets, A_799, b_800, n_801, p_1833);
    IntList_44 PREST_1832 = TAIL(pl_797);
    loop_2111(PREST_1832, IPROD(DTOI(p_1833), P_798), A_799, b_800, n_801,
              B_802, &_2117, &_2118, &_2119, &_2120, &_2121);
    if (ISNIL(_2121)) {
      *_2112 = CONS(_2093, _2117);
      *_2113 = CONS(p_1833, _2118);
      *_2114 = CONS(P_798, _2119);
      *_2115 = CONS(MDINV(p_1833, MDHOM(p_1833, P_798)), _2120);
      *_2116 = PREST_1832;
      return;
    } else {
      *_2112 = CONS(_2093, _2117);
      *_2113 = CONS(p_1833, _2118);
      *_2114 = CONS(P_798, _2119);
      *_2115 = CONS(MDINV(p_1833, MDHOM(p_1833, P_798)), _2120);
      *_2116 = _2121;
      return;
    }
  } else {
    *_2112 = NIL;
    *_2113 = NIL;
    *_2114 = NIL;
    *_2115 = NIL;
    *_2116 = NIL;
    return;
  }
}

static inline TUPLE_794 loop_796(IntList_44 pl_797, Integer P_798,
  IntMatrix_110 A_799, IntegerList_107 b_800, INT n_801, Integer B_802)
{
  IntList_44 _2112;
  IntList_44 _2113;
  IntegerList_107 _2114;
  IntList_44 _2115;
  LIST_778 _2116;
  loop_2111(pl_797, P_798, A_799, b_800, n_801, B_802, &_2112, &_2113,
    &_2114, &_2115, &_2116);
  return (TUPLE5(_2112, _2113, _2114, _2115, _2116));
}

static inline INT nth_583(INT n_584, IntList_44 l_585)
{
  while (!ISNIL(l_585))
    if (n_584 == 1)
      return (HEAD(l_585));
    else {
      l_585 = TAIL(l_585);
      n_584 = (n_584 - 1);
    }
  ERROR("nth");
}
```

```
static inline Integer loop_1277(IntList_44 _1278, IntList_44 _1279,
  IntegerList_107 _1280, LIST_778 _1281, Integer _1282, INT _1283)
{
  while (!ISNIL(_1278))
    if (ISNIL(_1279))
      return (_1282);
    else if (ISNIL(_1280))
      return (_1282);
    else if (ISNIL(_1281))
      return (_1282);
    else {
      TASK_777 a_2074 = HEAD(_1281);
      LIST_778 at_2073 = TAIL(_1281);
      Integer M_2072 = HEAD(_1280);
      IntegerList_107 Mt_2071 = TAIL(_1280);
      INT mi_2070 = HEAD(_1279);
      IntList_44 mit_2069 = TAIL(_1279);
      INT m_2068 = HEAD(_1278);
      IntList_44 mt_2067 = TAIL(_1278);
      DigitPair_114 _2094 = WAIT(a_2074);
      if (ELEM1(_2094) != 0) {
        _1282 = MIDCRA(M_2072, m_2068, mi_2070, _1282, nth_583(_1283,
          ELEM2(_2094)));
        _1281 = at_2073;
        _1280 = Mt_2071;
        _1279 = mit_2069;
        _1278 = mt_2067;
      } else
        return (_1282);
    }
  return (_1282);
}

static inline Integer fun_1033(INT _1034, LIST_778 _1035, IntList_44 _1036,
  IntegerList_107 _1037, IntList_44 _1038)
{
  return (loop_1277(_1036, _1038, _1037, _1035, DTOI(1), _1034));
}

static inline TASK_936 linSolve_956(IntMatrix_110 A_957,
  IntegerList_107 b_958, INT n_959, IntList_44 PLIST_960)
{
  while (TRUE) {
    LIST_778 _2126;
    IntList_44 _2127;
    IntegerList_107 _2128;
    IntList_44 _2129;
    IntList_44 _2130;
    TASK_936 _2095;
    loop_2111(PLIST_960, DTOI(1), A_957, b_958, n_959,
      IPROD(DTOI(2), IPROD(IFACTL(n_959), IEXP(loop_622(A_957,
      loop_616(b_958, DTOI(0))), n_959))), &_2126, &_2127, &_2128, &_2129,
      &_2130);
    _2095 = START5(fun_929, n_959, _2126, _2127, _2128, _2129);
    if (loop_937(_2126))
      return (_2095);
    else
      PLIST_960 = _2130;
  }
}

static inline void fun_2131(LIST_840 _1415, TASK_839 _1416, LIST_740 * _2132)
{
  while (TRUE) {
    LIST_840 _1531 = TAIL(_1415);
    if (ISNIL(_1531)) {
      *_2132 = NIL;
      return;
    } else {
```

```
      TASK_739 _2096 = START2(fun_1160, _1531, _1416);
      _2132 = RCONS(_2096, _2132);
      _1415 = _1531;
    }
  }
}

static inline LIST_740 fun_1414(LIST_840 _1415, TASK_839 _1416)
{
  LIST_740 _2132;
  fun_2131(_1415, _1416, &_2132);
  return (_2132);
}

static inline LIST_740 fun_929(INT n_930, LIST_778 dkyk_931,
  IntList_44 pk_932, IntegerList_107 Pk_933, IntList_44 ik_934)
{
  TASK_892 _2099 = START5(fun_885, n_930, dkyk_931, pk_932, Pk_933, ik_934);
  TASK_839 _2098 = START4(fun_904, dkyk_931, pk_932, Pk_933, ik_934);
  LIST_840 _2100 = WAIT(_2099);
  if (ISNIL(_2100))
    return (NIL);
  else {
    TASK_739 _2097 = START2(fun_1160, _2100, _2098);
    return (CONS(_2097, fun_1414(_2100, _2098)));
  }
}

static inline void loop_2134(INT _873, INT _874, LIST_778 _875,
  IntList_44 _876, IntegerList_107 _877, IntList_44 _878, LIST_840 * _2135)
{
  while (_873 <= _874) {
    TASK_839 _2101 = START5(fun_1033, _873, _875, _876, _877, _878);
    _2135 = RCONS(_2101, _2135);
    _873 = (_873 + 1);
  }
  *_2135 = NIL;
  return;
}

static inline LIST_840 loop_872(INT _873, INT _874, LIST_778 _875,
  IntList_44 _876, IntegerList_107 _877, IntList_44 _878)
{
  LIST_840 _2135;
  loop_2134(_873, _874, _875, _876, _877, _878, &_2135);
  return (_2135);
}

static inline LIST_840 fun_885(INT n_886, LIST_778 dkyk_887,
  IntList_44 pk_888, IntegerList_107 Pk_889, IntList_44 ik_890)
{
  return (loop_872(1, n_886, dkyk_887, pk_888, Pk_889, ik_890));
}

/*
 * external entry function
 */

RatVector_111 linSolve(IntMatrix_110 A, IntegerList_107 b, INT n)
{
  LIST_740 _2087 = WAIT(linSolve_956(A, b, n, PRIME));
  return (listsync_764(_2087));
}

/*
 * End of file
 */
```

# B Runtime System Interface

```
/******************************************************************************
 *
 * dc.h
 * runtime system interface of DC compiler
 *
 * Copyright (C) 1993 Wolfgang Schreiner, see file COPYRIGHT.
 *
 *****************************************************************************/

/* $Id: extern.tex,v 1.1 1994/06/18 19:09:18 ws Exp $ */

#ifndef DC_H
#define DC_H

#if !defined(DC_INLINE) || !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
 ******************************************************************************/

#define NO_SACLIB_MACROS
#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)

/* pseudo-function for integer to digit conversion, only if ISATOM(a)! */
__inline__
static Word ITOD(Word a) {
  return(a);
}
/* pseudo-function for digit to integer conversion */
__inline__
static Word DTOI(Word a) {
  return(a);
}

#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 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
 ******************************************************************************/

#define NO_SACLIB_MACROS
#include "paclib.h"

#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) pacAbort("D2C Error %s\n", s);

__inline__
static Word _COMP(Word a, Word L) {
  WORD M = pacNew();
  SFIRST(M,a);
  SRED(M,L);
  return(M);
}

__inline__
static Word *_RCOMP(Word a, Word *L) {
  WORD M = pacNew();
  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)

/* pseudo-function for integer to digit conversion, only if ISATOM(a)! */
__inline__
static Word ITOD(Word a) {
  return(a);
}
/* pseudo-function for digit to integer conversion */
__inline__
static Word DTOI(Word a) {
  return(a);
}

#define START0(f) pacStart(f,0)
#define START1(f,a) pacStart(f,1,a)
#define START2(f,a,b) pacStart(f,2,a,b)
#define START3(f,a,b,c) pacStart(f,3,a,b,c)
#define START4(f,a,b,c,d) pacStart(f,4,a,b,c,d)
#define START5(f,a,b,c,d,e) pacStart(f,5,a,b,c,d,e)
#define START6(f,a,b,c,d,e,g) pacStart(f,6,a,b,c,d,e,g)
#define START7(f,a,b,c,d,e,g,h) pacStart(f,7,a,b,c,d,e,g,h)
#define START8(f,a,b,c,d,e,g,h,i) pacStart(f,8,a,b,c,d,e,g,h,i)

#define FSTART0(r,f) pacFStart(r,f,0)
#define FSTART1(r,f,a) pacFStart(r,f,1,a)
#define FSTART2(r,f,a,b) pacFStart(r,f,2,a,b)
#define FSTART3(r,f,a,b,c) pacFStart(r,f,3,a,b,c)
#define FSTART4(r,f,a,b,c,d) pacFStart(r,f,4,a,b,c,d)
#define FSTART5(r,f,a,b,c,d,e) pacFStart(r,f,5,a,b,c,d,e)
#define FSTART6(r,f,a,b,c,d,e,g) pacFStart(r,f,6,a,b,c,d,e,g)
#define FSTART7(r,f,a,b,c,d,e,g,h) pacFStart(r,f,7,a,b,c,d,e,g,h)
#define FSTART8(r,f,a,b,c,d,e,g,h,i) pacFStart(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

#endif

/*****************************************************************************
 * end of dc.h
 *****************************************************************************/
```

# C Lexical and Parser Syntax

In this section, we give the lexical and the parser syntax of the para-functional D language in lex (actually GNU Flex) and yacc (actually GNU Bison) notation.

```
/*****************************************************************************
 *
 * lexical syntax
 *
 *****************************************************************************/
```

```
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); }

/******************************************************************************
 *
 * parser syntax
 *
 ******************************************************************************/

/* 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
;
```