Sized Types for Typing Eden Skeletons*

Ricardo Peña and Clara Segura

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid
e-mail: {ricardo,csegura}@sip.ucm.es

Abstract. The parallel-functional language Eden extends Haskell with constructs to explicitly define and communicate processes. These extensions allow the easy definition of skeletons as higher-order functions. However, the programmer can inadvertently introduce busy loops or deadlocks in them. In this paper a sized type system is extended in order to use it for Eden programs, so that those well-typed skeletons are guaranteed either to terminate or to be productive. The problems raised by Eden features and their possible solutions are described in detail, and several skeletons are manually type checked in this modified system such as the parallel map, farm, pipeline, and replicated workers.

1 Introduction

The parallel-functional language Eden [BLOP98] extends the lazy functional language Haskell with constructs to explicitly define and communicate processes. It is implemented by modifying the *Glasgow Haskell Compiler* (GHC) [PHH+93]. The three main additional concepts are *process abstractions*, *process instantiations* and the non-deterministic process abstraction `merge`. Process abstractions of type `Process a b` can be compared to functions of type `a -> b`, and process instantiations can be compared to function applications, the main difference being that the former, when instantiated, are executed in parallel. An instantiation is achieved by using the predefined infix operator `(#) :: Process a b -> a -> b`. Each time an expression `e1 # e2` is evaluated, a new parallel process is created to evaluate `(e1 e2)`. Non-determinism is introduced in Eden by means of a predefined process abstraction `merge :: Process [[a]] [a]` which *fairly* interleaves a set of input lists, to produce a single non-deterministic list.

These extensions allow the easy definition of *skeletons* as higher-order functions [PR01]. A skeleton [Col89] is a generic scheme for solving in parallel a particular family of problems. They are very useful because they are defined once and reused many times. However the programmer can inadvertently introduce busy loops, deadlocks or any other runtime error in them. So, it is very important to formally verify that they are free from these undesirable problems. The theory of sized types has been developed in recent years by John Hughes and Lars Pareto [HPS96,Par97,Par00] to provide a framework in which type checking based analysis of both program termination and program productivity can be done. A simplified version of Haskell, *Synchronous Haskell*, is given a

* Work partially supported by the Spanish-British Acción Integrada HB 1999-0102 and Spanish project TIC 2000-0738.

sized type system so that well-typed programs are guaranteed to be free from runtime errors.

The objective of this paper is to prove Eden skeletons correct. As they are not many, for the moment we will be glad with type checking them manually. In the way of conjecturing and proving their types correct, we have found some weaknesses in Hughes and Pareto's system. Thus, another objective is to propose extensions to the sized type system so that it could more be useful for typing Eden programs. The problems due to Eden's features and their possible solutions are described in detail, and several skeletons are type ckecked in this modified system.

The plan of the paper is as follows: Section 2 describes the theory of sized types developed by Hughes and Pareto and the type system for Synchronous Haskell. In Section 3 some problems introduced by Eden features are described and the type rules are extended consequently. Two simple examples, a naïve version of a parallel map skeleton and a pipeline are type checked using the new rules. In Section 4 more complex skeletons, such as the farm skeleton and the replicated workers topology are type checked. The problems posed by these skeletons and their possible solutions are discussed. In Section 5 we draw some conclusions and future work.

2 Huges and Pareto's Sized Types

From a semantic point of view, the denotation of a sized type is an upwards closed subset of a lattice which may not include $\perp$, where $\perp$ means at the same time non-termination (for finite types) and deadlock (for infinite types). So, if a function can be successfully typed in this system, it is sure that either it terminates (if the function produces a finite value), or it is productive (if it produces an infinite value). To this purpose, finite types must be carefully distinguished from infinite ones. Additionally, types may have one or more *size parameters* which carry size information. These can be constants, universally quantified variables or, in general (restricted) expressions. Intuitively, the size of a value of a certain type is the number of constructor applications needed to construct the value. For instance, for a finite list, it is the number of *cons* constructors plus one (the latter takes into account the *nil* constructor). For a binary tree, it is the number of levels plus one, and so on.

2.1 The Syntax of Sized Types and Datatype Terms

Syntactically, finite non-recursive types are introduced by a **data** declaration, finite recursive types by an **idata** declaration, and infinite ones by a **codata** declaration. Figure 1 shows the syntax of signatures and type declarations. There, τ, σ, s, k and t respectively denote types, type schemes, size expressions, size variables and type variables. A size expression can be either finite, denoted i, or infinite, denoted ω. In the former case, notice that they are restricted to be linear natural number expressions in a set of size variables. Size and type variables can be universally quantified in a type scheme and in a type declaration. There are some additional restrictions of well-formedness (e.g. that constructors must

$\tau ::= t \mid \tau \to \tau \mid T\ \overline{s}\ \overline{\tau}$	$D ::= \mathbf{data}\ E \mid \mathbf{idata}\ E \mid \mathbf{codata}\ E$
$\sigma ::= \forall t.\tau \mid \forall k.\tau \mid \tau$	$E ::= L = R \mid \forall t.E \mid \forall k.E$
$s ::= w \mid i$	$L ::= T\ \overline{s}\ \overline{\tau}$
$i ::= k \mid n \mid p * i \mid i + i$	$R ::= c_1\ \overline{\tau_1} \mid \ldots \mid c_n\ \overline{\tau_n}$

Fig. 1. Syntax of syzed types and datatype definitions

be unique, or that mutually recursive definitions are not allowed), all of which can be statically checked, in order that types have a well defined semantics. See [Par00] for details. Examples of valid type declarations are:

data `Bool` $=$ `true` | `false`, **idata** $\forall a$. `List` $\omega\ a =$ `nil` | `cons` a (`List` $\omega\ a$),
idata `Nat` $\omega =$ `zero` | `succ` (`Nat` ω), **codata** $\forall a$. `Strm` $\omega\ a =$ `make` a (`Strm` $\omega\ a$)

representing respectively, the boolean type, finite lists of any size, natural numbers of any size, and streams of any size. Examples of valid type schemes are: `List` 3 (`Nat` ω), $(a \to b) \to$ `Strm` $k\ a \to$ `Strm` $k\ b$, $\forall a.\forall k$. `List` $k\ a \to$ `Nat` k.

2.2 Semantics of Sized Types

The first parameter of an **idata** or a **codata** type is a size expression bounding the size of the values of that type. For **idata** it is an *upper bound*, and for **codata** it is a *lower bound*. So, the type **List** `2 a` denotes lists of zero or one value (i.e. having at most two constructors), while **Stream** `2 a` denotes streams of two or more values (i.e. having at least two `make` constructors). Partial streams such as $1; 2; \bot$ and infinite streams such as $1; 2; \ldots$ (where ; is the infix version of `make`) belong to this type. This size parameter can be instantiated with the infinite size ω. For **idata** types such as **List** ω `a` this means finite lists of *any* size, while for **codata** types such as **Stream** ω `a` this means strictly *infinite* streams.

A program consists of a set of well-formed type declarations followed by a term written in an enriched λ-calculus having constructor applications, **case** expressions, and **letrec** expressions, each one with a set of mutually recursive simple bindings. In order to preserve type soundness, there is also the restriction that all constructors of the corresponding type must appear exactly once in the branches of every **case** expression.

The universe of values $\mathbf{U}$ is defined as the solution to the following isomorphism: $\mathbf{U} \sim [\mathbf{U} \to \mathbf{U}]_\bot \oplus (\mathbf{U} \times \mathbf{U})_\bot \oplus \mathbf{1}_\bot \oplus CON_\bot$, where CON is the set of constructors. In this universe, the enriched λ-calculus is given a standard non-strict semantics. The set of types $\mathbf{T} = \{\mathcal{T} \mid \mathcal{T}$ is an upwards closed subset of $\mathbf{U}\}$ form a complete lattice under the subset ordering $\subseteq$ where $\top_{\mathbf{T}} = \mathbf{U}$ is the top element, $\bot_{\mathbf{T}} = \emptyset$ is the bottom element, and $\cup$, $\cap$ are respectively the least upper bound and greatest lower bound operators. It is a cpo but not a domain. As types are upwards closed subsets of $\mathbf{U}$, the only type containing $\bot_{\mathbf{U}}$ is just $\mathbf{U}$. In this cpo, recursive types are interpreted by using functionals $\mathcal{F} : \mathbf{T} \to \mathbf{T}$. To define these functionals from type declarations, in [Par00] several type operators, $\boxed{\times}$ for non-strict cartesian products of types, $\boxed{+}$ for sums of types, and $\boxed{\to}$ for functions between types, are defined. It is proved that these operators preserve the upwards closedness property.

From now on, we will use an overline to represent several elements. For example, $\overline{s}$ represents that there exists some number $l \geq 0$ such that $s = s_1 \ldots s_l$. The denotation of a recursive type constructor instantiation $T\ \overline{s}\ \overline{\tau}$ is the application of a function that takes as parameters $l-1$ naturals corresponding to the sizes $s_2, \ldots, s_l$, j types corresponding to the types $\tau_1, \ldots, \tau_j$, and returns a *type iterator* $\mathcal{F} : \mathbf{T} \to \mathbf{T}$. If all declarations respect the forementioned static semantics restrictions, the resulting type iterators are continuous for **idata** declarations and co-continuous for **codata** ones. This type iterator takes as parameter a type corresponding to the recursive ocurrences of the type being defined, and returns a new type corresponding to its right hand side definition. The first size parameter s_1 determines the number of times the iterator is applied (respectively to $\bot_{\mathbf{T}}$ for **idata** or to $\top_{\mathbf{T}}$ for **codata**). If it is applied k times, then values of at most (at least, for **codata**) size k are obtained. In the limit (size ω), values of any size (infinite size, for **codata**) are obtained. For inductive types (those defined by an **idata** declaration) the interpretation is the least fixpoint of the iterator, while for co-inductive types (those defined by a **codata** declaration) it is the greatest fixpoint of the iterator. These fixpoints can respectively be reached by the limits of the ascending chain $\mathcal{F}^i(\bot_{\mathbf{T}})$ and of the descending chain $\mathcal{F}^i(\top_{\mathbf{T}})$.

A subtype relation can be defined between sized types with the same underlying type but different size. It is a subset based relation, that is, $\tau \rhd \tau'$ when the interpretation of τ is a subset of the interpretation of τ'. It is a monotone relation with the sizes in **idata** types, while antimonotone in the **codata** types. For example $\mathbf{List}\ 2\ a \rhd \mathbf{List}\ 3\ a$, while $\mathbf{Strm}\ 3\ a \rhd \mathbf{Strm}\ 2\ a$. In [Par00] the rules for checking this subtyping relation are shown. They provide a way of weakening size information. This weakening will be applied in [*APP*], [*LET*], [*LETREC*] and [*CASE*] rules, see Figure 4.

The interpretation of type polymorphism is, as usual, the intersection of the interpretations of every single type. In formal terms, if $\mathcal{I}$ denotes the interpretation function, γ is a type environment mapping type variables to types in $\mathbf{T}$, and δ is a size environment mapping size variables to sizes in $\mathbb{N}^\omega$, then $\mathcal{I}[\![\forall t.\sigma]\!]\ \gamma\ \delta = \bigcap_{\mathcal{T} \in \mathbf{T}} \mathcal{I}[\![\sigma]\!]\ \gamma[\mathcal{T}/t]\ \delta$. Similarly, size polymorphism is interpreted as the intersection of the interpretation for all sizes: $\mathcal{I}[\![\forall k.\sigma]\!]\ \gamma\ \delta = \bigcap_{n \in \mathbb{N}} \mathcal{I}[\![\sigma]\!]\ \gamma\ \delta[n/k]$; but in this case the intersection is restricted to finite sizes. The reason for that is to be able to use induction on natural numbers to assign types to recursive definitions. This restriction makes size instantiation of polymorphic types safe only for finite sizes. *Omega instantiation* is not always safe. Indeed, there are some types for which omega instantiation delivers a type *smaller* than the polymorphic one instead of bigger. An example is $\forall k.(\mathbf{Stream}\ \omega\ (Nat\ k) \to Bool)$ substituting ω for k. In [Par00], a sufficient decidable condition (namely that a type scheme is *undershooting* with respect to a size variable, denoted as $\sigma \overset{\cup}{\sim} k$) is given for a scheme to be able to be safely instantiated with ω.

2.3 Synchronous Haskell Type System

In [Par00] the programming language Synchronous Haskell is defined. This is a simplified version of the functional language Haskell. Thanks to its demand

driven evaluation it can be seen as a concurrent language where programs are visualized as networks of processes executing in parallel and communicating through message channels.

A program consists of a set of type declarations followed by a term written in an enriched λ-calculus. Each **let** and **letrec** binding is annotated by the programmer with a type scheme to be (type) checked by the system. The whole type system is not shown here as our extended type rules in Figures 4 are just an extension of these. To recover the original type system, first eliminate [*PABS*], [*PINST*] and [*MERGE*]. Then, eliminate the classes F and T in the type schemes and the environments Γ_T^T and Γ_T^F in the type assertions. Finally, eliminate all those conditions where $\overline{t}^F$, $\overline{t}^T$, Γ_T^T, Γ_T^F or any of their variants are involved. So the type assertions are of the form $\Gamma \vdash e :: \tau$. In this section we only explain [*VAR*] and [*LETREC*]. Different indexes will be used when different sequences of elements appear in the same rule. We will not use i because it is reserved for finite size expressions. So for example in [*VAR*], $\tau \overset{\cup}{\sim} k'_j$ means that being $\overline{k'} = k'_1 \ldots k'_n$, then $\forall j \in \{1..n\}.\tau \overset{\cup}{\sim} k'_j$. There, instantiation of both type and size variables takes place. We can only instantiate with ω the size variables k'_j such that τ is undershooting w.r.t. them. In [Par00] sufficient conditions are given to prove this property, which imply the definition of other relations between types and size variables, such as monotonicity $\tau \overset{+}{\sim} k$ and antimonotonicity $\tau \overset{-}{\sim} k$.

In [*LETREC*], induction on natural numbers is applied. To illustrate how this rule works we will explain the typing of function $map :: \forall a, b, k.(a \to b) \to \mathbf{List}\ k\ a \to \mathbf{List}\ k\ b$, where $map = \lambda f.\lambda xs.\mathbf{case}\ xs\ \mathbf{of}\ nil \to nil; cons\ x\ xs' \to cons\ (f\ x)\ (map\ f\ xs')$. First of all we have to choose the size variable on which the induction is made. In the rule it is always the first one k_j in each binding. Then the base case is studied $\sigma_j[0] = \mathbf{U}$. This is called the *bottom check*. The rules for checking it need the definition of other relations on types as emptiness ($= \mathbf{E}$) and non-emptiness ($\neq \mathbf{E}$); and some other on sizes, like $= 0$ and $\neq 0$. All the rules are in [Par00], but we show the most useful ones in Figure 2. In general, a **codata** type (T_C) of size 0 denotes the universe, while a **idata** type (T_I) of size 0 denotes the empty type. Some examples where the bottom check holds are: $\forall k.\mathbf{Strm}\ k\ a$, $\forall a, b, k.(a \to b) \to \mathbf{List}\ k\ a \to \mathbf{List}\ k\ b$ and $\forall a, k.\mathbf{Strm}\ k\ a \to \mathbf{Strm}\ k\ a$. Then, assuming the induction hypothesis for each $j \in \{1..n\}$: $x_j :: \forall \overline{k_j}.\tau_j[k]$ we must prove that the types of the right hand sides of the bindings e_j are subtypes of $\tau_j[k+1]$. In the example, assuming that $map :: (a \to b) \to \mathbf{List}\ k\ a \to \mathbf{List}\ k\ b$ we must prove that its body definition has type $(a \to b) \to \mathbf{List}\ (k+1)\ a \to \mathbf{List}\ (k+1)\ b$. This implies $f :: a \to b$ and $xs :: \mathbf{List}\ (k+1)\ a$. In the **case** expression, if xs is an empty list, $nil :: \mathbf{List}\ 1\ a$ is returned, which is a subtype of $\mathbf{List}\ (k+1)\ a$ as $1 \leq k+1$. If it is non-empty, the value is destroyed, so xs' has one constructor less $xs' :: \mathbf{List}\ k\ a$. By induction hypothesis then $map\ f\ xs' :: \mathbf{List}\ k\ b$, so adding a new element gives us $cons\ (f\ x)\ (map\ f\ xs') :: \mathbf{List}\ (k+1)\ b$, the desired type. Polymorphic recursion in all the size variables but the inductive one is allowed, and it is quite useful, for example to type the *reverse* function, as shown in [Par00].

$$\frac{\tau_2 = \mathbf{U}}{\tau_1 \to \tau_2 = \mathbf{U}} \qquad \frac{\tau_1 = \mathbf{E}}{\tau_1 \to \tau_2 = \mathbf{U}} \qquad \frac{s_1 = 0}{T_C\ \overline{s}\ \overline{\tau} = \mathbf{U}} \qquad \frac{\sigma = \mathbf{U}}{\forall t.\sigma = \mathbf{U}} \qquad \frac{\sigma = \mathbf{U}}{\forall k.\sigma = \mathbf{U}} \qquad \frac{s_1 = 0}{T_I\ \overline{s}\ \overline{\tau} = \mathbf{E}}$$

Fig. 2. Rules for bottom check and one rule for emptiness check

3 New Problems Introduced by Eden

Eden is a parallel functional language where processes communicate through channels, so it is very close to the view Synchronous Haskell provides to programs. However, some problems arise when trying to apply Synchronous Haskell type system to Eden programs. If we take into account Eden features, some extensions to the type system are needed. The most important features are the way in which values are transmitted through the channels, the eager evaluation of some expressions, and the use of lists to represent both Haskell lists and stream-like transmission of values.

The instantiation protocol of `e1 # e2` deserves some attention in order to understand Eden's semantics: (1) closure `e1` together with all its dependent closures are *copied* unevaluated to a new processor and the child process is created there to evaluate it; (2) once created, the child process starts producing eagerly its output expression; (3) expression `e2` is eagerly evaluated in the parent process. If it is a tuple, an independent concurrent thread is created to evaluate each component (we will refer to each tuple element as a *channel*). Once a process is running, only fully evaluated data objects are communicated. The only exception are lists: they are transmitted in a *stream*-like fashion, i.e. element by element. Each list element is first evaluated to normal form and then transmitted.

3.1 Transmission of Values

The communication of data through channels leads us to two different problems. Firstly, as values are evaluated to normal form before sending them, it is necessary that the types of the communicated values are finite: tuple, **data** or **idata** types with finite components and function types. Secondly, this implies that instantiation of type variables must be restricted in some places to finite types. We propose a mechanism similar to the class system of Haskell. We define a *finiteness relation* $\Gamma_T \vdash_F \tau$, see Figure 3, where Γ_T is a set of type variables that may appear in τ. This assertion means that assuming the type variables in Γ_T can only be instantiated with finite types, τ is also a finite type. In Figure 3 we use a pseudo-variable F to substitute a finite type for the recursive positions of an **idata** type, that is, to prove by structural induction that it is finite.

We use this relation to control the instantiation of variables. As there are different ways of sending values, depending on the channel type, we can use different types to represent a channel: if it is a single value channel, its type is represented by the type of the value; if it is a stream-like channel, we can use a **List** or a **Strm** type (see discussion below). This separation imposes different restrictions: if it is a single value channel, its type must be finite, but if it is a stream-like channel, only the type of the elements must be finite. This leads us to introduce two different classes of values, F (from finite) and T (from

$$\frac{a \in \Gamma_T}{\Gamma_T \vdash_F a} \qquad \frac{\Gamma_T \vdash_F \tau_1 \qquad \Gamma_T \vdash_F \tau_2}{\Gamma_T \vdash_F (\tau_1, \tau_2)} \qquad \frac{}{\Gamma_T \vdash_F \tau \to \tau'} \qquad \frac{}{\Gamma_T \vdash_F F}$$

$$\frac{\Gamma_T \vdash_F R[\overline{\tau}/\overline{t}][\overline{s}/\overline{k}]}{\Gamma_T \vdash_F T_d\ \overline{s}\ \overline{\tau}} \qquad \frac{\Gamma_T \vdash_F S[F][\overline{\tau}/\overline{t}][\overline{s}/\overline{k}]}{\Gamma_T \vdash_F T_i\ s_1\ \overline{s}\ \overline{\tau}} \qquad \frac{\forall j \in \{1..n\}, l \in \{1..m_j\} \quad \Gamma_T \vdash_F \tau_{jl}}{\Gamma_T \vdash_F c_1\ \overline{\tau_1} \mid \ldots \mid c_n\ \overline{\tau_n}}$$

where **data** $\forall \overline{k}\ \overline{t}.T_d\ \overline{k}\ \overline{t} = R$ $\quad R = c_1\ \overline{\tau_1} \mid \ldots \mid c_n\ \overline{\tau_n}$
idata $\forall \overline{k}\ \overline{t}.T_i\ w\ \overline{k}\ \overline{t} = S[T_i\ w\ \overline{k}\ \overline{t}]$ $\quad S = c_1\ \overline{\tau_1} \mid \ldots \mid c_n\ \overline{\tau_n}$

Fig. 3. Finiteness relation

transmission interface). The first one indicates that the type variable can only be instantiated with finite types, and the second one that it can only be instantiated with '*interface*' types. An interface type is either a finite type (this includes the **List** type with finite components) or a **Strm** with finite components. A process usually has several input and output channels, represented by a tuple of channels, so interface types must also include tuples of the two previous types. The types are extended with these classes: $\tau' ::= \tau \mid [T\ \overline{a}], [F\ \overline{b}] \Rightarrow \tau$; and a new rule for bottom check is needed, expressing that it is not affected by the contexts:

$$\frac{\tau = \mathbf{U}}{[T\ \overline{a}], [F\ \overline{b}] \Rightarrow \tau = \mathbf{U}}$$

Additionally, two new environments, Γ_T^F and Γ_T^T are introduced in the rules, carrying the type variables that appear respectively in a F or a T context. So our assertions are of the form $\Gamma_T^F, \Gamma_T^T, \Gamma \vdash e :: \tau$. The predicate $P(\Gamma_T^F, \Gamma_T^T, \tau)$, defined in Figure 4 tells us whether τ is a interface type.

In Figure 4 the modified type rules are shown. We describe here only the newly introduced elements. In [*VAR*] the instantiation of type variables is controlled: those that appear in an F context are instantiated with finite types, and those that appear in a T context are instantiated with a interface type. In a **Process** $\tau\ \tau'$ type, τ and τ' represent the communication interfaces, so in [*PABS*] and [*PINST*] it must be checked that they are in fact interface types. In [*MERGE*], the values transmitted through the channels must be of finite type. We use angle brackets there to represent strict tuples, explained in Section 4.2.

In [*LET*] ([*LETREC*] is similar), the bindings are annotated with their types. If a universally quantified type variable is qualified by a class F or T, we force the programmer to indicate the same quantification in all the annotations where such variable appears free ($\overline{t^T} \subseteq \Gamma_T^T$ and $\overline{t^F} \subseteq \Gamma_T^F$). Additionally, the class information needed to type the right hand side of a binding is extracted from its annotation ($\Gamma_T^{T\prime} = \overline{b}$ and $\Gamma_T^{F\prime} = \overline{c}$). The rest of rules ($\lambda$-abstraction, application and **case**) are similar to the original ones.

3.2 Eager Evaluation

In Eden, lazy evaluation is changed to eager in two cases: (1) processes are eagerly instantiated when the expression under evaluation demands the creation of a closure of the form $o = e_1 \# e_2$, and (2) instantiated processes produce their output even if it is not demanded. These semantics modifications are aimed at

$$\frac{\begin{array}{c}\sigma = \forall \overline{a}\ \overline{k}\ \overline{k'}.T\ \overline{b}, F\ \overline{c} \Rightarrow \tau \\ \overline{t^T} = \overline{a} \cap \overline{b} \qquad \overline{t^F} = \overline{a} \cap \overline{c} \qquad \overline{t} = \overline{a} \backslash (\overline{t^T} \cup \overline{t^F}) \\ \tau \overset{\cup}{\sim} k'_j \qquad \Gamma_T^F \vdash_F \tau_m^F \qquad P(\Gamma_T^F, \Gamma_T^T, \tau_l^T)\end{array}}{\Gamma_T^F,\ \Gamma_T^T,\ \Gamma \cup \{x :: \sigma\} \vdash x :: \tau[\overline{i}/\overline{k}][\overline{s}/\overline{k'}][\overline{\tau}/\overline{t}][\overline{\tau^T}/\overline{t^T}][\overline{\tau^F}/\overline{t^F}]}\ \mathit{VAR}$$

$$\frac{\Gamma_T^F, \Gamma_T^T, \Gamma \cup \{x :: \tau\} \vdash e :: \tau' \quad P(\Gamma_T^F, \Gamma_T^T, \tau) \quad P(\Gamma_T^F, \Gamma_T^T, \tau')}{\Gamma_T^T, \Gamma_T^T, \Gamma \vdash \mathbf{process}\ x \to e :: \mathbf{Process}\ \tau\ \tau'}\ \mathit{PABS}$$

$$\frac{\begin{array}{c}\Gamma_T^F, \Gamma_T^T, \Gamma \vdash e_1 :: \mathbf{Process}\ \tau\ \tau' \qquad \Gamma_T^F, \Gamma_T^T, \Gamma \vdash e_2 :: \tau'' \qquad \tau'' \rhd \tau \\ P(\Gamma_T^F, \Gamma_T^T, \tau) \qquad P(\Gamma_T^F, \Gamma_T^T, \tau')\end{array}}{\Gamma_T^F, \Gamma_T^T, \Gamma \vdash e_1 \# e_2 :: \tau'}\ \mathit{PINST}$$

$$\frac{\Gamma_T^F, \Gamma_T^T, \Gamma \vdash e :: \langle \mathbf{Strm}\ k_1\ \tau, \ldots, \mathbf{Strm}\ k_n\ \tau \rangle \quad \Gamma_T^F \vdash_F \tau}{\Gamma_T^F, \Gamma_T^T, \Gamma \vdash merge \# e :: \mathbf{Strm}\ (\sum_{j=1}^{n}\ k_j)\ \tau}\ \mathit{MERGE}$$

$$\frac{\begin{array}{c}\sigma = \forall \overline{a}\ \overline{k}.T\ \overline{b}, F\ \overline{c} \Rightarrow \tau \quad \overline{a}, \overline{k} \notin FV(\Gamma) \\ \overline{t^T} = \overline{b} \backslash \overline{a} \qquad \overline{t^F} = \overline{c} \backslash \overline{a} \qquad \Gamma_T^{T'} = \overline{b} \qquad \Gamma_T^{F'} = \overline{c} \\ \Gamma_T^F, \Gamma_T^T, \Gamma \cup \{x :: \sigma\} \vdash e' :: \tau' \quad \Gamma_T^{F'}, \Gamma_T^{T'}, \Gamma \vdash e :: \tau'' \quad \tau'' \rhd \tau \\ \overline{t^T} \subseteq \Gamma_T^T \qquad \overline{t^F} \subseteq \Gamma_T^F\end{array}}{\Gamma_T^F, \Gamma_T^T, \Gamma \vdash \mathbf{let}\ x :: \sigma = e\ \mathbf{in}\ e' :: \tau'}\ \mathit{LET}$$

$$\frac{\begin{array}{c}j \in \{1..n\}, \sigma_j[k_j] = \forall \overline{a_j}\ \overline{k_j}.T\ \overline{b_j}, F\ \overline{c_j} \Rightarrow \tau_j[k_j] \quad \overline{a_j}, \overline{k_j}, k_j \notin FV(\Gamma) \\ \Gamma' = \Gamma, x_1 :: \forall \overline{k_1}.\tau_1[k] \ldots x_n :: \forall \overline{k_n}.\tau_n[k] \\ \Gamma'' = \Gamma, x_1 :: \forall k_1.\sigma_1[k_1] \ldots x_n :: \forall k_n.\sigma_n[k_n] \\ \Gamma_T^{T'} = \overline{b_1} \cup \ldots \cup \overline{b_n} \qquad \Gamma_T^{F'} = \overline{c_1} \cup \ldots \cup \overline{c_n} \qquad \overline{t} = \overline{a_1} \cup \ldots \cup \overline{a_n} \\ \overline{t^T} = \Gamma_T^T \backslash \overline{t} \qquad \overline{t^F} = \Gamma_T^F \backslash \overline{t} \qquad \overline{t^T} \subseteq \Gamma_T^T \qquad \overline{t^F} \subseteq \Gamma_T^F \\ \sigma_j[0] = \mathbf{U} \qquad \Gamma_T^{F'}, \Gamma_T^{T'}, \Gamma' \vdash e_j :: \tau'_j \qquad \tau'_j \rhd \tau_j[k+1] \qquad \Gamma_T^F, \Gamma_T^T, \Gamma'' \vdash e :: \tau\end{array}}{\Gamma_T^F, \Gamma_T^T, \Gamma \vdash \mathbf{letrec}\ x_1 :: \forall k_1.\sigma_1[k_1] = e_1 \ldots x_n :: \forall k_n.\sigma_n[k_n] = e_n\ \mathbf{in}\ e :: \tau}\ \mathit{LETREC}$$

$$\frac{\Gamma_T^F, \Gamma_T^T, \Gamma \cup \{x :: \tau_1\} \vdash e_: : \tau_2}{\Gamma_T^F, \Gamma_T^T, \Gamma \vdash \lambda x.e :: \tau_1 \to \tau_2}\ \mathit{ABS}$$

$$\frac{\Gamma_T^F, \Gamma_T^T, \Gamma \vdash e_1 :: \tau_1 \to \tau_2 \qquad \Gamma_T^F, \Gamma_T^T, \Gamma \vdash e_2 :: \tau_3 \quad \tau_3 \rhd \tau_1}{\Gamma_T^F, \Gamma_T^T, \Gamma \vdash e_1\ e_2 :: \tau_2}\ \mathit{APP}$$

$$\frac{\begin{array}{c}\vdash_= \tau = c_1\ \overline{\tau_1} \mid \ldots \mid c_n\ \overline{\tau_n} \\ \Gamma_T^F, \Gamma_T^T, \Gamma \vdash e :: \tau \qquad \Gamma_T^F, \Gamma_T^T, \Gamma \cup \{x_{j1} :: \tau_{j1}, \ldots, x_{jn_j} :: \tau_{jn_j}\} \vdash e_j :: \tau'_j(\forall j) \qquad \tau'_j \rhd \tau'(\forall j)\end{array}}{\Gamma_T^F, \Gamma_T^T, \Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ c_1\ \overline{x_1} \to e_1 \ldots c_n\ \overline{x_n} \to e_n :: \tau'}\ \mathit{CASE}$$

$$\begin{array}{l} P(\Gamma_T^F, \Gamma_T^T, \tau) = \mathit{if}\ (\tau = \mathbf{Strm}\ s\ \tau')\ \mathit{then}\ \Gamma_T^F \vdash_F \tau' \\ \qquad\qquad \mathit{else\ if}\ (\tau = (\tau_1, \ldots, \tau_n)\ \mathit{then}\ \forall j. \\ \qquad\qquad\qquad \mathit{if}\ \tau_j = \mathbf{Strm}\ s_j\ \tau'_j\ \mathit{then}\ \Gamma_T^F \vdash_F \tau'_j \\ \qquad\qquad\qquad \mathit{else} \quad \Gamma_T^F \cup \Gamma_T^T \vdash_F \tau_j \\ \qquad\qquad \mathit{else} \quad \Gamma_T^F \cup \Gamma_T^T \vdash_F \tau \end{array}$$

Fig. 4. Type rules

increasing the degree of parallelism and at speeding up the distribution of the computation.

Eager evaluation does not affect the type rules. From the type system point of view, eagerness means that some values of interface types, like o in $o = e_1 \# e_2$, are produced without being demanded. If o is finite, its type gives us an upper bound of its size. With lazy evaluation, this size needs not be reached in all cases, while with eager evaluation, this size is probably reached all times the expression $e_1 \# e_2$ is evaluated. If o is a stream, its type gives a lower bound of the number of elements produced, provided there is demand for them. With eager evaluation the only difference is that this demand is guaranteed.

3.3 Types List and Strm

In Eden, the list type $[\tau]$ is used both for Haskell lists and for stream-like channels, and they are transformed from one to the other in a way transparent to the programmer. However, in this type system it is necessary first to divide Haskell lists into finite ones (**List** type) and partial or infinite ones (**Strm** type). This is a problem inherited from Haskell. A **List** type gives us a proof of termination, while a **Strm** type gives us a proof of productivity. Additionally it is necessary to identify the stream-like channels. We usually want to prove the productivity of our skeletons, so we will use mainly the **Strm** type in such cases. But there are some skeletons that work with finite types and require a version with **List** of another skeleton. In such cases we would like to have both versions of the skeleton, one for lists and another one for streams, so that both termination and productivity are proved. In some cases we obtain the two versions for free, thanks to polymorphism, as in the naïve map and pipe skeletons shown below.

3.4 Two Simple Examples

We study now two simple examples of skeletons that illustrate some of the ideas shown in this section. In the following section we will study more complex skeletons and the problems they produce. In all of them, we first show the Eden skeleton as it is written in [PR01] and then the sized typed version appears. The latter is usually modified somehow for different reasons we will explain in turn. Similarly to Hughes and Pareto's system there is not an automatic way of transforming the programs (so that they are easier to type) or of conjecturing a correct type in a first attempt.

In order to abbreviate type proofs, in those functions and skeletons where, to type them, induction has been used, we will write as a subscript the size of those program variables whose type contains the size variable over which we are doing induction. Sometimes we will write the size of a complete expression. When a compound type is used, as in **List** k (**Strm** l a), we will use brackets to represent the size, in the example $k[l]$. In the original text [PR01] a `Transmissible` class (abbreviated `Tr` here) is used. It subsumes both the T and F classes used in the type system.

A Naïve Implementation of a Parallel Map Skeleton We first show a naïve implementation of a map skeleton:

```
map_naive :: (Tr a, Tr b) => (a -> b) -> [a] -> [b]
map_naive f xs = [pf # x | x <- xs]    where pf = process x -> f x
```

For each element of the list, a different process instantiation is done, where each process simply applies the function `f` to the corresponding input.

The ZF notation is rewritten into a simple map and the **where** clause into a **let.** The type is obtained by composing functions:

$$\begin{aligned} &\mathit{map\_naiveL} :: \forall a, b, k.T\ a, b \Rightarrow (a \to b) \to \mathbf{List}\ k\ a \to \mathbf{List}\ k\ b \\ &\mathit{map\_naiveL} = \lambda f.\lambda xs.\mathbf{let}\ g :: T\ a, b \Rightarrow a \to b \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad g = \lambda x.(\mathbf{process}\ y \to f\ y)\#x \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\mathbf{in}\ \mathit{map}\ g\ xs \end{aligned}$$

A Pipeline Skeleton Now we show a pipeline skeleton instantiating a different process to evaluate each of the pipeline stages. Each process in the pipe creates its successor process:

```
pipe :: Tr a => [[a] -> [a]] -> [a] -> [a]
pipe fs xs  = (ppipe fs) # xs
ppipe :: Tr a => [[a] -> [a]] -> Process [a] [a]
ppipe [f]    = process xs -> f xs
ppipe (f:fs) = process xs -> (ppipe fs) # (f xs)
```

The following type can be checked by induction on the length k of fs:

$$\begin{aligned} &\mathit{ppipe} :: \forall a, k, l.F\ a \Rightarrow \mathbf{List}\ k\ (\mathbf{Strm}\ l\ a \to \mathbf{Strm}\ l\ a) \to \mathbf{Process}\ (\mathbf{Strm}\ l\ a)\ (\mathbf{Strm}\ l\ a) \\ &\mathit{ppipe} = \lambda fs_{k+1}.\mathbf{case}\ fs_{k+1}\ \mathbf{of} \\ &\qquad\qquad \mathit{nil} \qquad\quad \to \mathbf{process}\ s \to s \\ &\qquad\qquad \mathit{cons}\ f\ {fs'}_k \to \mathbf{process}\ s \to (\mathit{ppipe}\ {fs'}_k)\#(f\ s) \end{aligned}$$

$$\begin{aligned} &\mathit{pipe} :: \forall a, k, l.F\ a \Rightarrow \mathbf{List}\ k\ (\mathbf{Strm}\ l\ a \to \mathbf{Strm}\ l\ a) \to \mathbf{Strm}\ l\ a \to \mathbf{Strm}\ l\ a \\ &\mathit{pipe} = \lambda fs.\lambda s.(\mathit{ppipe}\ fs)\#s \end{aligned}$$

In this example we have encountered and solved a couple of problems. First, the empty list case is not included in the original `ppipe`, which violates the restrictions for **case.** So it is added as the identity process in order to be able to type the skeleton.

Second, we have chosen to represent the processes as consuming and producing a stream of data in order to study its productivity. This means that the type a cannot be an interface type, but must be a finite type, so the context in this case is F and not T. However we could have given less restrictive types: $\mathit{ppipe} :: \forall a, k, l.T\ a \Rightarrow \mathbf{List}\ k\ (a \to a) \to \mathbf{Process}\ a\ a$ and $\mathit{pipe} :: \forall a, k, l.T\ a \Rightarrow \mathbf{List}\ k\ (a \to a) \to a \to a$, using T class. By instantiating a with $F\ a \Rightarrow \mathbf{Strm}\ l\ a$ we obtain the previous type.

4 Skeletons in Eden

4.1 The Farm Implementation of the Parallel Map Skeleton

The `map_naive` version can be improved by reducing the number of worker processes to be created. In a `map_farm` a process is created for every processor, tasks are evenly distributed between processors, and the results are collected. Here is its implementation in terms of `map_naive`:

```
map_farm :: (Tr a,Tr b) => (a -> b) -> [a] -> [b]
map_farm = farm noPe unshuffle shuffle
farm :: (Tr a, Tr b) => Int -> (Int->[a]->[[a]]) -> ([[b]]->[b]) ->
        (a -> b) -> [a] -> [b]
farm np unshuffle shuffle f tasks =
                        shuffle (map_naive (map f) (unshuffle np tasks))
```

where `noPe` is a constant giving the number of available processors. Different strategies to split the work into the different processes can be used provided that, for every list `xs`, `(shuffle . unshuffle) xs == xs`. For instance, the following scheme distributes the tasks using a round-robin strategy:

```
unshuffle :: Int -> [a] -> [[a]]
unshuffle n ins
   | length firsts < n  = take n (map (:[]) firsts ++ repeat [])
   | otherwise          = zipWith (:) firsts (unshuffle n rest)
  where (firsts, rest) = splitAt n ins
shuffle :: [[a]] -> [a]
shuffle = concat . transpose
```

In Figure 5 the typings of auxiliary functions used in the farm skeleton are shown. Function *zipWiths* (we only show its type), *takes*, *drops* and $(++)_s$ are proved by induction on k. Notice the use of the subtyping relation $\triangleright$. In Figure 6 the modified farm skeleton is shown with its type. There, functions $\mathit{unshuffles}^n$ and *shuffles* are proved by induction on l.

The first thing to decide is which list types are finite lists and which ones are streams. In order to study the productivity we have chosen stream types for the input `[a]` and output `[b]` lists in the skeleton. As the number of processes is finite, the distribution of tasks between processes is considered a list of streams. This decision leads us to slightly change the definitions of `shuffle` and `unshuffle`, so they have now a different type. In particular, `shuffle` needs an auxiliary stream in order to cope with a possibly empty list of channels, even though this situation will never arise in practice, as it would correspond to having zero processors.

Some problems have been found when typing this skeleton: The first one arises when typing `unshuffle`. We are dividing a stream of elements into n lists of streams. This means that the original stream should have at least $n * k$ elements, so that we obtain n lists of at least k elements. This implies a product of size variables, which is not allowed by the type system. There are two possibilities to solve this. One is to define a family $\mathit{unshuffles}^n$ of functions, one for each fixed number of processes n, so that $n*k$ is a product of a constant and a variable. This means that the natural parameter would disappear, and consequently $farm^n$ and map_farm^n would also be a family of functions. The drawback of this alternative is that we need to define many versions of the same skeleton for several values of n. The other possibility is to allow products of two (or more) variables, or size expressions, in order to obtain a parametric skeleton. In this case some new rules (not shown) for checking the relations $= 0$, $\neq 0$, $\overset{+}{\sim} k$ and $\overset{-}{\sim} k$ should be added. The drawback of this alternative is that products of variables are not included in the type checking algorithm, and they may even make it undecidable.

$$
\begin{array}{l}
\textbf{sidata sList}\ w\ a = \mathit{snil} \mid \mathit{scons}\ a\ (\textbf{sList}\ w\ a) \\
\mathit{zipWiths} :: \forall a,b,c,k.(a \to b \to c) \to \textbf{sList}\ k\ a \to \textbf{sList}\ k\ b \to \textbf{sList}\ k\ c \\
\mathit{takes} :: \forall a,k,l.\mathit{Nat}\ k \to \textbf{Strm}\ (k+l)\ a \to \textbf{sList}\ k\ a \\
\mathit{takes} = \lambda n_{k+1}.\lambda s.\textbf{case}\ n_{k+1}\ \textbf{of}\ \mathit{zero} \to \mathit{snil}_{1 \rhd k+1} \\
\qquad\qquad\qquad\qquad\qquad \mathit{succ}\ n'_k \to \textbf{case}\ s_{k+l+1}\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad x; s'_{k+l} \to (\mathit{scons}\ x\ (\mathit{takes}\ n'_k\ s'_{k+l}))_{k+1} \\
\mathit{drops} :: \forall a,k,l.\mathit{Nat}\ k \to \textbf{Strm}\ (k+l)\ a \to \textbf{Strm}\ l\ a \\
\mathit{drops} = \lambda n_{k+1}.\lambda s_{k+l+1}.\textbf{case}\ n_{k+1}\ \textbf{of}\ \mathit{zero} \to s_{k+l+1 \rhd l} \\
\qquad\qquad\qquad\qquad\qquad \mathit{succ}\ n'_k \to \textbf{case}\ s_{k+l+1}\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad x; s'_{k+l} \to (\mathit{drops}\ n'_k\ s'_{k+l})_l \\
\mathit{splitAts} :: \forall a,k,l.\mathit{Nat}\ k \to \textbf{Strm}\ (k+l)\ a \to (\textbf{sList}\ k\ a, \textbf{Strm}\ l\ a) \\
\mathit{splitAts} = \lambda n.\lambda s.(\mathit{takes}\ n\ s, \mathit{drops}\ n\ s) \\
(+\!+_s) :: \forall a,k,l.\textbf{sList}\ k\ a \to \textbf{Strm}\ l\ a \to \textbf{Strm}\ l\ a \\
xs_{k+1}\ +\!+_s\ s = \textbf{case}\ xs_{k+1}\ \textbf{of}\ \mathit{snil} \to s_l;\quad \mathit{scons}\ x\ xs'_k \to (x; (xs'_k\ +\!+_s\ s)_l)_{l+1 \rhd l}
\end{array}
$$

Fig. 5. Auxiliary functions for the farm skeleton

The second problem is also related to `unshuffle`. Its resulting type is a list of streams. The bottom check fails when $l = 0$, as $\textbf{List}\ n\ \mathbf{U} \neq \mathbf{U}$. This problem arises when we use a list or a tuple to represent several channels coming out of a process. As a simpler example with tuples we show the following version of *unshuffle* for two streams:

$$
\begin{array}{l}
\mathit{unshuffle2} :: \forall a,k.\textbf{Strm}\ (2k)\ a \to \langle \textbf{Strm}\ k\ a, \textbf{Strm}\ k\ a \rangle \\
\mathit{unshuffle2} = \lambda s_{2(k+1)}.\textbf{case}\ s_{2(k+1)}\ \textbf{of} \\
\qquad\qquad x; s'_{2k+1} \to \textbf{case}\ s'_{2k+1}\ \textbf{of} \\
\qquad\qquad\quad y; s''_{2k} \to \textbf{let}\ \langle s1_k, s2_k \rangle = \mathit{unshuffle2}\ s''_{2k}\ \textbf{in}\ \ ((x; s_1)_{k+1}, (y; s_2)_{k+1})
\end{array}
$$

A similar problem arised in [Par00] when trying to type mutually recursive definitions. This was solved by building a special fixpoint definition for a set of simultaneous equations. A function from tuples to tuples would not work in principle as the bottom check fails because $(\mathbf{U}, \mathbf{U}) \neq \mathbf{U}$. The solution we propose is to untag the tuples, that is, to define strict tuples. We define strict data, **sdata** (T_s), and idata, **sidata** (T_{si}). We add a new rule to represent the strictness, and another one to establish when an **sidata** is empty:

$$
\frac{\exists i.\tau_i = \mathbf{U}}{T_s/T_{si}\ \overline{s}\ \overline{\tau} = \mathbf{U}} \qquad\qquad \frac{s_1 = 0}{T_{si}\ \overline{s}\ \overline{\tau} = E}
$$

From now on angle brackets will represent strict tuples. These were also used in rule [*MERGE*], see Figure 4. Strict lists are defined in Figure 5. In order to make the previous strictness rule semantically correct we define a new type operator $\boxed{\times'}$ used to interpret this kind of types: $\mathcal{T}_1 \boxed{\times'} \mathcal{T}_2 = \begin{cases} \mathcal{T}_1 \boxed{\times} \mathcal{T}_2 \text{ if } \mathcal{T}_1, \mathcal{T}_2 \neq \mathbf{U} \\ \mathbf{U} \text{ otherwise} \end{cases}$

4.2 Replicated Workers Topology

We now show a replicated workers implementation [KPR00] of the parallel map skeleton that distributes work on demand, i.e. a new task is assigned to a process only if it is known that it has already finished its previous work. The programmer cannot predict in advance the order in which processes are going to finish their works, as this depends on runtime issues.

$$
\begin{array}{l}
farm^n :: \forall a,b,l.F\ a,b \Rightarrow (\mathbf{Strm}\ (n*l)\ a \rightarrow \mathbf{sList}\ n\ (\mathbf{Strm}\ l\ a)) \rightarrow \\
\qquad\qquad (\mathbf{sList}\ n\ (\mathbf{Strm}\ l\ a) \rightarrow \mathbf{Strm}\ l\ b \rightarrow \mathbf{Strm}\ l\ b) \rightarrow \\
\qquad\qquad (a \rightarrow b) \rightarrow \mathbf{Strm}\ l\ b \rightarrow \mathbf{Strm}\ (n*l)\ a \rightarrow \mathbf{Strm}\ l\ b \\
farm^n = \lambda unshuffle^n.\lambda shuffle.\lambda f.\lambda aux.\lambda s.shuffle\ (map\_naiveLs\ (mapS\ f)\ (unshuffle^n\ s))\ aux
\end{array}
$$

$$
\begin{array}{l}
map\_farm^n :: \forall a,b,k.F\ a,b \Rightarrow (a \rightarrow b) \rightarrow \mathbf{Strm}\ k\ b \rightarrow \mathbf{Strm}\ (n*k)\ a \rightarrow \mathbf{Strm}\ k\ b \\
map\_farm^n = farm^n\ unshuffles^n\ shuffles
\end{array}
$$

$$
\begin{array}{l}
unshuffles^n :: \forall a,l.\mathbf{Strm}\ (n*l)\ a \rightarrow \mathbf{sList}\ n\ (\mathbf{Strm}\ l\ a) \\
unshuffles^n = \lambda s_{n(l+1)}.\mathbf{let}\ (firsts_n, rest_{nl}) = splitAts\ n\ s_{n(l+1)} \\
\qquad\qquad \mathbf{in}\ (zipWith\ (;)\ firsts_n\ (unshuffles^n\ rest_{nl})_{n[l]})_{n[l+1]}
\end{array}
$$

$$
\begin{array}{l}
shuffles :: \forall a,l,k.\mathbf{sList}\ (k+1)\ (\mathbf{Strm}\ l\ a) \rightarrow \mathbf{Strm}\ l\ b \rightarrow \mathbf{Strm}\ l\ b \\
shuffles = \lambda xs_{k+1[l+1]}.\lambda aux_{l+1}.\mathbf{case}\ xs_{k+1[l+1]}\ \mathbf{of} \\
\qquad\qquad snil \rightarrow aux_{l+1} \\
\qquad\qquad scons\ s_{l+1}\ {xs'}_{k[l+1]} \rightarrow \mathbf{case}\ s_{l+1}\ \mathbf{of} \\
\qquad\qquad\qquad\qquad x; {s'}_l \rightarrow \mathbf{let}\ heads_k = map\ hdS\ {xs'}_{k[l+1]} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in\ let}\ tls_{k+1[l]} = map\ tlS\ xs_{k+1[l+1]} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ (x; (heads_k +\!\!+_s (shuffles\ tls_{k+1[l]}\ aux)_l)_l)_{l+1}
\end{array}
$$

Fig. 6. The farm skeleton

By using the reactive (and non-deterministic) process `merge`, acknowledgments from different processes can be received by the manager as soon as they are produced. Thus, if each acknowledgment contains the identity of the sender process, the list of merged results can be scrutinized to know who has sent the first message, and a new work can be assigned to it:

```
rw :: (Tr a, Tr b) => Int -> Int -> (a->b) -> [a] -> [b]
rw np prefetch fw tasks =   results    where
  results = sortMerge outputsChildren
  outputsChildren =  [(worker fw i) # inputs
                        |(i,inputs) <- zip [0..np-1] inputss]
  inputss         = distribute tasksAndIds
                        (initReqs ++ (map owner unorderedResult))
  tasksAndIds     = zip [1..] tasks
  initReqs        = concat (generate prefetch [0..np-1])
  unorderedResult = merge # outputsChildren   -- Non-deterministic!!

  distribute [] _          = generate np []
  distribute (e:es) (i:is) = insert i e (distribute es is)
    where insert 0 e ~(x:xs) = (e:x):xs
          insert (n+1) e ~(x:xs) = x:(insert n e xs)

worker :: (Tr a, Tr b) => (a->b) -> Int -> Process [(Int,a)]  [ACK b]
worker f i = process ts -> map (\(id_t,t) -> ACK i id_t (f t)) ts

data ACK b =  ACK Int Int b
owner (ACK i _ _ )  = i
```

The skeleton receives as input parameters (1) the number of worker processes to be used; (2) the size of workers' prefetching buffer; (3) the worker function that will perform the actual computation on the tasks; and (4) the list of tasks into

$$
\begin{array}{l}
\textbf{data}\ ACK\ k\ b = ACK\ (Nat\ k)\ b \\
generate :: \forall k.Nat\ k \to \textbf{Strm}\ k\ (Nat\ k) \\
generate = \lambda n_{k+1}.\textbf{case}\ n_{k+1}\ \textbf{of}\ zero \to zeros_{\omega[1]\triangleright k+1[k+1]}; \\
\qquad\qquad\qquad\qquad succ\ n'_k \to (n_{k+1}; (generate\ n'_k)_{k[k]})_{k+1[k+1]} \\
\\
zipS :: \forall a,b,k.\textbf{Strm}\ k\ a \to \textbf{Strm}\ k\ b \to \textbf{Strm}\ k\ (a,b) \\
zipS = \lambda s_{k+1}.\lambda t_{k+1}.\textbf{case}\ s_{k+1}\ \textbf{of}\ x; s'_k \to \textbf{case}\ t_{k+1}\ \textbf{of}\ y; t'_k \to ((x,y); (zipS\ s'_k\ t'_k)_k)_{k+1} \\
\\
owner :: \forall b,k.ACK\ k\ b \to Nat\ k \\
owner\ (ACK\ i\ \_) = i \\
result :: \forall b,k.ACK\ k\ b \to b \\
result\ (ACK\ \_\ b) = b \\
\\
worker :: \forall a,b,k,l.F\ a,b \Rightarrow (a \to b) \to Nat\ k \to \textbf{Process}\ (\textbf{Strm}\ l\ a)\ (\textbf{Strm}\ l\ (ACK\ k\ b)) \\
worker = \lambda f.\lambda n.\textbf{process}\ ts \to \textbf{let}\ f' :: F\ a,b \Rightarrow a \to ACK\ k\ b \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad f' = \lambda t.ACK\ n\ (f\ t) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ (mapS\ f'\ ts) \\
\\
mapt^n :: \forall a,b,k_1,\ldots,k_n.F\ a,b \Rightarrow (a \to b) \to \langle \textbf{Strm}\ k_1\ a,\ldots,\textbf{Strm}\ k_n\ a\rangle \to \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \langle \textbf{Strm}\ k_1\ (ACK\ n\ b),\ldots,\textbf{Strm}\ k_n\ (ACK\ n\ b)\rangle \\
mapt^n = \lambda w.\lambda\langle s_1,\ldots,s_n\rangle.\langle (worker\ w\ 0)\#s_1,\ldots,(worker\ w\ (n-1))\#s_n\rangle
\end{array}
$$

Fig. 7. Auxiliary functions for the replicated workers topology

which the problem has been split. See [KPR00] for details. In Figure 7 the types of auxiliary functions used in this topology are shown. Functions *generate* and *zipS* are proved by induction on k. Notice that we make use of subtyping in *generate*. We are assuming that $mapS :: \forall a,b,k.(a \to b) \to \textbf{Strm}\ k\ a \to \textbf{Strm}\ k\ b$.

In Figure 8 a modified version of the topology is given a type. We have simplified some aspects. The prefetch parameter has been eliminated. The task identity has also been eliminated from the *ACK* type so we eliminate the sorting function and just return the unordered result. Several problems have been found when typing this skeleton. The first one is the following. As this is a topology where work is distributed on demand, the sizes of the streams communicating the processes are not necessarily the same, so working with lists of streams is not appropriate, as too much information would be lost. So we have decided to use instead strict tuples of streams of different sizes: $\langle \textbf{Strm}\ k_1\ a,\ldots,\textbf{Strm}\ k_n\ a\rangle$. This means that the topology is in fact a family of functions, one for a different number of processors. So the natural number parameter is eliminated.

The second problem is that typing the recursive **let** implies proving that all the streams grow at least in one element, and this is not true for `outputChildren` for example. However it can be noticed that once `distribute` is applied, at least one task will be given to a process in `inputss` so that the topology keeps running. Then, we redefine the recursive **let** by defining a single recursive binding for *inputss* in terms of the rest of the bindings. So it is only necessary to prove that *inputss* grows. This is true at the beginning of the execution thanks to the initial list of requests *initReqs*. The rest of requests are appended after these: `initReqs ++ map owner unorderedResult`.

The third problem arises in this append function. We have to append a finite list of length n to a stream of at least, say, k elements. In the type system

List n a means that the list has a length of at most n, so we cannot safely say that the resulting stream has at least $n+k$ elements. We need to use a stream also for the first parameter. This is *generate* n in Figure 8. It generates the first n requests and then adds an infinite number of 0's at the end. So, given two streams, the new append function, takes n elements from the first one and puts them at the beginning of the second one. But again this is not inmediate, because we have to take n elements from the first stream, and again the natural numbers are inductive. So, we need to define a (inductively defined) family of append functions $\{{+\!+}^j :: \forall a,k.\mathbf{Strm}\ j\ a \to \mathbf{Strm}\ k\ a \to \mathbf{Strm}\ (k+j)\ a\}_{j\in\mathbb{N}^+}$, where now each function is defined in terms of the previous one:

$$
\begin{array}{l}
{+\!+}^1 :: \forall a,k.\mathbf{Strm}\ 1\ a \to \mathbf{Strm}\ k\ a \to \mathbf{Strm}\ (k+1)\ a \\
s\ {+\!+}^1\ s' = \mathbf{case}\ s\ \mathbf{of}\ x;s'' \to x;s'
\end{array}
$$

$$
\begin{array}{l}
{+\!+}^{j+1} :: \mathbf{Strm}\ (j+1)\ a \to \mathbf{Strm}\ k\ a \to \mathbf{Strm}\ (k+j+1)\ a \\
s\ {+\!+}^{j+1}\ s' = \mathbf{case}\ s\ \mathbf{of}\ x;s'' \to x;(s''\ {+\!+}^j\ s')
\end{array}
$$

The last problem arises in function `distribute`. This function provides new tasks to idle processes. This means that the streams are not uniformly increased. So it is not possible to type the *distribute*^{n} function of Figure 8 as we do not know which stream will be increased. But we are sure that one of them will be. We need then to make *induction over the sum of the stream sizes* and not separately on one of them. This is not supported by the type system. The process would be the following. First the bottom check: $\sum_{j=1}^{n} k_j = 0$ implies that for each $j = 1,\ldots,n$, $k_j = 0$. This means that $\mathbf{Strm}\ k_j\ a = \mathbf{U}$ and the use of strict tuples leads us to $\langle \mathbf{Strm}\ k_1\ a,\ldots,\mathbf{Strm}\ k_n\ a\rangle = \mathbf{U}$. The induction hypothesis is that $distribute^n :: \mathbf{Strm}\ w\ a \to \mathbf{Strm}(\sum_{j=1}^{n} k_j)\ (Nat\ n) \to \langle \mathbf{Strm}\ k_1\ a,\ldots,\mathbf{Strm}\ k_n\ a\rangle$ where $\sum_{j=1}^{n} k_j = k$. We have to prove that $distribute^n :: \mathbf{Strm}\ w\ a \to \mathbf{Strm}(\sum_{j=1}^{n} k'_j)$ $(Nat\ n) \to \langle \mathbf{Strm}\ k'_1\ a,\ldots,\mathbf{Strm}\ k'_n\ a\rangle$ where $k+1 = \sum_{j=1}^{n} k'_j = \sum_{j=1}^{n} k_j + 1$. So let $s_1 :: \mathbf{Strm}\ w\ a$ and $s_2 :: \mathbf{Strm}\ (\sum_{j=1}^{n} k_j + 1)\ (Nat\ n)$. Then, by applying the **case** rule, $s'_1 :: \mathbf{Strm}\ w\ a$ and $s'_2 :: \mathbf{Strm}\ (\sum_{j=1}^{n} k_j)\ (Nat\ n)$, so applying the induction hypothesis we obtain that $distribute^n\ s'_1\ s'_2 :: \langle \mathbf{Strm}\ k_1\ a,\ldots,\mathbf{Strm}\ k_n\ a\rangle$; and then, for each $j = 1,\ldots,n$, $t_j :: \mathbf{Strm}\ k_j\ a$. In each branch of the **if** expression one of the components is increased by one element, that is, each branch has a type where the sum of sizes is $k+1$, so the whole expression has a type where the sum of sizes is $k+1$. This is what we wanted to prove. This kind of induction should be also applied to *inputss*. This is still not included in the extended type system and should be formalised as a new way of induction.

5 Conclusions and Future Work

Eden is a parallel functional language that is sufficiently expressive that the programmer may introduce deadlocks and non-termination in programs. Having a proof that these unwanted effects are not present is highly desirable. Such proofs are even more essential for skeletons because the latter represent parallel topologies reused many times in different applications.

We have extended sized types by Hughes and Pareto in several directions in order to make them more useful for typing our skeletons, and proved that the

$$
\begin{array}{l}
rw^n :: \forall a,b.F\ a,b \Rightarrow (a \to b) \to \mathbf{Strm}\ w\ a \to \mathbf{Strm}\ w\ b \\
rw^n = \lambda f.\lambda tasks. \\
\quad \mathbf{let}\ initReqs :: \mathbf{Strm}\ n\ (Nat\ n) \\
\quad\quad initReqs = generate\ n \\
\quad \mathbf{in\ let\ rec} \\
\quad\quad inputss :: \forall k_1 \ldots k_n.F\ a \Rightarrow \langle \mathbf{Strm}\ k_1\ a, \ldots, \mathbf{Strm}\ k_n\ a \rangle \\
\quad\quad inputss = \mathbf{let}\ \ oChildren :: F\ b \Rightarrow \langle \mathbf{Strm}\ k_1\ (ACK\ n\ b), \ldots, \mathbf{Strm}\ k_n\ (ACK\ n\ b) \rangle \\
\quad\quad\quad\quad\quad\quad oChildren = mapt^n\ f\ inputss \\
\quad\quad\quad\quad\quad\quad \mathbf{in\ let}\ unordered :: F\ b \Rightarrow \mathbf{Strm}\ (\sum_{i=1}^{n} k_i)\ (ACK\ n\ b) \\
\quad\quad\quad\quad\quad\quad\quad\quad unordered = merge\#oChildren \\
\quad\quad\quad\quad\quad\quad \mathbf{in\ let}\ restReqs :: \mathbf{Strm}\ (\sum_{i=1}^{n} k_i)\ (Nat\ n) \\
\quad\quad\quad\quad\quad\quad\quad\quad restReqs = mapS\ owner\ unordered \\
\quad\quad\quad\quad\quad\quad \mathbf{in\ let}\ requests :: \mathbf{Strm}\ (\sum_{i=1}^{n} k_i + n)\ (Nat\ n) \\
\quad\quad\quad\quad\quad\quad\quad\quad requests = initReqs{+\!\!+}^n restReqs \\
\quad\quad\quad\quad\quad\quad \mathbf{in}\ distribute^n\ tasks\ requests \\
\quad \mathbf{in\ let} \\
\quad\quad outputChildren :: \forall k_1 \ldots k_n.F\ b \Rightarrow \langle \mathbf{Strm}\ k_1\ (ACK\ n\ b), \ldots, \mathbf{Strm}\ k_n\ (ACK\ n\ b) \rangle \\
\quad\quad outputChildren = mapt^n\ f\ inputss \\
\quad \mathbf{in}\ mapS\ result\ (merge\#outputChildren)
\end{array}
$$

$$
\begin{array}{l}
distribute^n :: \forall a, k_1, \ldots k_n.\mathbf{Strm}\ w\ a \to \mathbf{Strm}(\sum_{k=1}^{n} k_i)\ (Nat\ n) \to \langle \mathbf{Strm}\ k_1\ a, \ldots, \mathbf{Strm}\ k_n\ a \rangle \\
distribute^n = \lambda s_1.\lambda s_2.\mathbf{case}\ s_1\ \mathbf{of}\ x_1; s_1' \to \mathbf{case}\ s_2\ \mathbf{of}\ x_2; s_2' \to \\
\qquad\qquad\qquad\qquad \mathbf{case}\ (distribute^n\ s_1'\ s_2')\ \mathbf{of}\ \langle t_1, \ldots, t_n \rangle \to \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{if}\ (x_2 == 0)\ then\ \langle x_1; t_1, \ldots, t_n \rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \{-(x_2 == n-1)-\}\ \langle t_1, \ldots, x_1; t_n \rangle
\end{array}
$$

Fig. 8. The replicated workers topology

skeletons are free from abnormal termination and busy loops. Firstly, we have introduced type classes for restricting the instantiation of some type variables. In this way, it can be proved that only finite values are communicated between processes. Secondly, we have added to the system strict types, and their corresponding typing rules, in order to be able to do induction proofs for processes receiving or/and producing tuples or finite lists of channels. Thirdly, we suggest to incorporate products of size variables in size expressions. This would make the type system much more expressive so that many more algorithms could be typed. In the general case the extension would probably make type checking undecidable, but surely many useful subcases could be encountered in which the algorithm is still decidable. Our `map_farm` and `map_rw` skeletons provide real examples showing that this extension is clearly needed. Finally, we propose to extend the induction proof embedded in rule [*LETREC*] to cope with doing induction on expressions such as the sum of size variables rather than only on single variables.

Even though the proposed extensions can automate the proofs, the programmer is still responsible for conjecturing a type for his/her program and for reformulating it in order to get a proof. We have typed other skeletons not shown here such as different versions of divide and conquer.

There are other techniques to prove termination and productivity of programs. Between the first ones they may be cited the big amount of work done in term rewriting systems and, in the functional field, proposals such as that of David Turner [Tur95], were all programs terminate by construction. On produc-

tivity, it may be cited [Sij89]. In Pareto's thesis [Par00] more exhaustive related work can be found.

As future work we plan in the first place to continue trying the typing of other (more complex) skeletons such as the ring and torus. We would also like to work in better formalizing the new features and in extending the current implementation of sized types with them in order to check automatically the proofs presented in this paper.

References

[BLOP98] S. Breitinger, R. Loogen, Y. Ortega Mallén, and R. Peña. Eden: Language Definition and Operational Semantics. Technical Report, Bericht 96-10. Revised version 1.998, Philipps-Universität Marburg, Germany, 1998.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research monographs in parallel and distributed computing. Pitman, 1989.

[HPS96] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT*, pages 410–423, 1996.

[KPR00] U. Klusik, R. Peña, and F. Rubio. Replicated Workers in Eden. 2nd International Workshop on Constructive Methods for Parallel Programming (CMPP 2000). To be published by Nova Science, 2000.

[Par97] L. Pareto. Sized types. Licenciate Dissertation, Chalmers University of Technology, Göteborg, Sweden, 1997.

[Par00] L. Pareto. *Types for Crash Prevention*. PhD thesis, Chalmers University of Technology and Göteborg University, Sweden, 2000.

[PHH+93] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain, and P. L. Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Inf. Technology, Keele*, pages 249–257, 1993.

[PR01] R. Peña and F. Rubio. Parallel Functional Programming al Two Levels of Abstraction. In *Principles and Practice of Declarative Programming, PPDP'01*. To appear in ACM Press, 2001.

[Sij89] B. A. Sijtsma. On the Productivity of Recursive List Definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, October 1989.

[Tur95] D. A. Turner. Elementary Strong Functional Programming. In *Functional Programming Languages in Education, FPLE'95*, pages 1–13. LNCS 1022, Springer, 1995.