Inferring Effect Types in an Applicative Language with Asynchronous Concurrency

Josva Kleist Martin Hansen
Bo Jensen Hans Hüttel

Department of Mathematics and Computer Science Aalborg University
Fredrik Bajersvej 7E
9220 Aalborg Ø
Denmark
Email: {kleist,mahans,bjens,hans}@iesd.auc.dk

November 21, 1996

Keywords: Type inference, process calculi, effect type, asynchrony.

1 Introduction

In recent years, a number of concurrent programming languages have been based around applicative languages. Best known are CML [Rep92], Facile [GMP89] and LCS [Ber93], all extensions of Standard ML [MTH90]. One would of course want to extend the type inference paradigm of Standard ML [Mil78] to cope with the new concurrency constructs.

Recently, the effect type discipline proposed by Lucassen [Luc87] and Talpin and Jouvelot [TJ92] has attracted a lot of interest as a possible means of capturing these non-applicative aspects, the idea being to associate with an expression not only its type but also the side effects of its evaluation.

Nielson and Nielson have proposed a monomorphic effect type discipline for CML [NN93]. Here the effect of executing a CML expression was a process calculus term. In [Tho93] Thomsen proposed a very different effect polymorphic effect type discipline for the Facile language, and independently, in [BD94] Bolignano and Debabi gave a very similar type discipline for a subset of Concurrent ML – here, the effect of a program is the *sort* of the program, i.e. the set of communication channels used and all causality is absent.

In this paper we consider λinda, a concurrent applicative language based on a rather different concurrency paradigm, namely the Linda concept of

Gelernter and Bernstein [GB82], and give a type system where effects are described by expressions in a process calculus.

The type inference system proposed in this paper introduces a novel combination of two previous solutions that have been used to deal with a problem peculiar to applicative programming languages, that of determining the polymorphic effect type of functions. Should one use subtyping or effect polymorphism? We argue that a combination is needed. We give a sound type reconstruction algorithm based around our idea.

Proofs of all results stated in the paper are found in [KJH94].

2 λinda

λinda adds the Linda concept of [GB82] to an applied call-by-value λ-calculus. Linda is based on the notion of a *tuple space*, a shared memory visible to all processes. A tuple space is a multiset of tuples – a tuple consists either of data or evaluating subprocesses. Each process in the program can modify the tuple space by depositing or removing tuples.

As tuples were simply a particular data structure introduced in the original implementation of Linda, we generalize the notion to that of *processes*. In λinda any meaningful expression can be deposited in a *process space* which is a multiset of expressions. Four primitives are used to manipulate the process space: in removes expressions, rd copies expressions from the process space. out deposits values and eval deposits expressions to be evaluated. Communication through a process space is *asynchronous* – processes only block if the expression they want to input does not exist in the process space. The choice of which expression to retrieve is made by *pattern matching*; λinda extends Linda in that patterns are first-class objects. Further, we extend λinda with multiple process spaces to gain the possibility of encapsulation.

2.1 Syntax

The syntax of λinda expressions is

$$\begin{array}{lcl} e & ::= & v \mid x \mid e_1\, e_2 \mid \ \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \ \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{rec}\, x(y)\cdot e \\ v & ::= & c \mid \lambda x\cdot e \mid <c\, v_1 \cdots v_n> \end{array}$$

Values are thus either constants, λ-abstractions or *closures*. The latter handle function constants – see below. Variables x range over the set **Var**. $\mathsf{rec}\, x(y)\cdot e$ defines a recursive function with parameter y and body e possibly containing recursive calls (occurrences of x).

The constants of λinda have the abstract syntax

$$\begin{array}{rcl} c & ::= & () \mid \mathsf{true} \mid \mathsf{false} \mid n \mid p \mid \mathsf{pair} \mid \mathsf{fst} \mid \mathsf{snd} \\ & \mid & \mathsf{nil} \mid \mathsf{cons} \mid \mathsf{head} \mid \mathsf{tail} \mid \mathsf{isnil} \mid ? \mid \mathsf{in} \mid \mathsf{rd} \mid \mathsf{out} \mid \mathsf{eval} \mid \mathsf{makeps} \end{array}$$

The set of constants includes the basic values: () of unit type, the booleans, integers $n \in \mathbb{Z}$, and process space identifiers $p \in$ **PS-id**. The latter cannot be written as literals but only appear as the result of creating a process space by makeps. The set of constants includes operators for making pairs and lists and the operators for communication: '?' is a pattern matching any value, in , rd, out and eval are the Linda primitives, and makeps is used to create new process spaces.

2.2 The Operational Semantics of λinda

The operational semantics of λinda has a sequential level dealing with the evaluation of a single expression and a concurrent level dealing with the evolution of the process spaces.

Sequential Evaluation

The semantics of the sequential evaluation is based on the notion of evaluation contexts of [Rep92]:

$$E \quad ::= \quad [\,] \mid E\,e \mid v\,E \mid \mathsf{let}\ x = E\ \mathsf{in}\ e \mid \mathsf{if}\ E\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$$

Any λinda expression is a completed evaluation context $E[e]$ containing the subexpression e to be evaluated; the definition of E ensures call-by-value semantics, c.f. the operational rules in Table 1. These define a labelled transition system with labels l denoting the communication occurring (if any):

$$l ::= \mathsf{in}(v_1, v_2, v_3) \mid \mathsf{rd}(v_1, v_2, v_3) \mid \mathsf{out}(v_1, v_2) \mid \mathsf{eval}(v_1, v_2\,()) \mid \mathsf{makeps}(v) \mid \varepsilon$$

with ε denoting internal computation.

The rule [app2] states that the application of two values (v_1, v_2) can evaluate to a third value v_3 if $(v_1, v_2, v_3) \in \Delta$. Δ, defined in Table 2, only deals with the first argument of the Linda primitives as application of the second argument has a side effect on the whole system and is handled by the rules for Linda primitives.

Concurrent Evaluation

A λinda program is a pool of process spaces. A process space $S \in$ **ProcSpace** is a multiset of expressions tagged with their type. $S[e{:}\tau]$ here denotes that

[app1]	$E[(\lambda x{\cdot}e)\, v] \xrightarrow{\varepsilon} E[e\{v/x\}]$	
[app2]	$E[v_1\, v_2\, \tau_2] \xrightarrow{\varepsilon} E[v_3]$	$\lvert if \rvert (v_1, v_2, v_3) \in \Delta$
[if1]	$E[\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2] \xrightarrow{\varepsilon} E[e_1]$	
[if2]	$E[\ \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2] \xrightarrow{\varepsilon} E[e_2]$	
[let]	$E[\ \mathsf{let}\ x = v\ \mathsf{in}\ e] \xrightarrow{\varepsilon} E[e\{v/x\}]$	
[rec]	$E[\mathsf{rec}\, x(y){\cdot}e] \xrightarrow{\varepsilon} E[(\lambda y{\cdot}e)\{(\mathsf{rec}\, x(y){\cdot}e)/x\}]$	
[in]	$E[<\ \mathsf{in}\ v_1 >\ v_2] \xrightarrow{\mathsf{in}(v_1,v_2,v_3)} E[v_3]$	
[rd]	$E[<\ \mathsf{rd}\ v_1 >\ v_2] \xrightarrow{\mathsf{rd}(v_1,v_2,v_3)} E[v_3]$	
[out]	$E[<\ \mathsf{out}\ v_1 >\ v_2] \xrightarrow{\mathsf{out}(v_1,v_2)} E[()]$	
[eval]	$E[<\ \mathsf{eval}\ v_1 >\ v_2] \xrightarrow{\mathsf{eval}(v_1,v_2\,())} E[()]$	
[makeps]	$E[\ \mathsf{makeps}\ ()] \xrightarrow{\mathsf{makeps}(v)} E[v]$	

Table 1: The semantics at the sequential level

$\Delta \supset \{$	$(\mathsf{pair},$	$v,$	$<\!\mathsf{pair}\, v\!>),$
	$(<\!\mathsf{pair}\, v_1\!>,$	$v_2,$	$<\!\mathsf{pair}\, v_1\, v_2\!>),$
	$(\mathsf{fst},$	$<\!\mathsf{pair}\, v_1\, v_2\!>,$	$v_1),$
	$(\mathsf{snd},$	$<\!\mathsf{pair}\, v_1\, v_2\!>,$	$v_2),$
	$(\ \mathsf{in}\ ,$	$v,$	$<\ \mathsf{in}\ \ v\!>),$
	$(\mathsf{eval},$	$v,$	$<\!\mathsf{eval}\, v\!>)$

Table 2: Clauses from the relation Δ dealing with evaluation of function constants

$e{:}\tau$ is an element of S. The pool of process spaces W maps process space identifiers to process spaces, i.e. $W \in$ **PS-id**$\hookrightarrow$**ProcSpace**.

The semantics of the concurrent evaluation is shown in Table 3. The evaluation of the Linda primitives within an expression results in a change to a process space. out and eval both result in the addition of an expression to a process space ([PSP-out] and [PSP-eval]). in removes a value if it matches the pattern ([PSP-in]), and the rd primitive acts as the in primitive except that the value is not removed ([PSP-rd]).

Observe that the rules [PSP-out] and [PSP-in] introduce an ordering of the changes: the process space containing the active expression is changed and only then is the process space being manipulated changed, as the two references may refer to the same process space.

Matching

Values are input or read from the process space using *pattern matching*; in λinda patterns are first-class objects and have the type τ pat. A pattern

$$\text{[PSP-elem]} \qquad \frac{e \xrightarrow{\varepsilon} e'}{W\,[p \mapsto S[e:\tau]] \longrightarrow W\,[p \mapsto S[e':\tau]]}$$

$$\text{[PSP-make]} \qquad \frac{e \xrightarrow{\mathsf{makeps}(p')} e' \quad p' \notin |dom|(W)}{W\,[p \mapsto S[e:\tau]] \longrightarrow W\,[p \mapsto S[e':\tau], p' \mapsto \emptyset]}$$

$$\text{[PSP-out]} \qquad \frac{e \xrightarrow{\mathsf{out}(p_2,v)} e'}{W\,[p_1 \mapsto S[e:\tau]] \longrightarrow W\,[p_2 \mapsto W'(p_2) \uplus \{\!\![v:\tau']\!\!\}]} \qquad \text{where } W'{=}W[p_1{\mapsto}S[E[e':\tau]]]$$

$$\text{[PSP-eval]} \qquad \frac{e \xrightarrow{\mathsf{eval}(p_2,e'')} e'}{W\,[p_1 \mapsto S[e:\tau]] \longrightarrow W\,[p_2 \mapsto W'(p_2) \uplus \{\!\![e'':\tau']\!\!\}]} \qquad \text{where } W'{=}W[p_1{\mapsto}S[e':\tau]]$$

$$\text{[PSP-in]} \qquad \frac{e \xrightarrow{\mathsf{in}(p_2,v_1,v_3)} e' \quad v_2:\tau_2 \in W(p_2) \quad v_3 = \mathsf{match}(v_1:\tau_1, v_2:\tau_2)}{W\,[p_1 \mapsto S[e:\tau]] \longrightarrow W\,[p_2 \mapsto W'(p_2) - \{\!\![v_2:\tau_2]\!\!\}]} \qquad \text{where } W'{=}W[p_1{\mapsto}S[e':\tau]]$$

$$\text{[PSP-rd]} \qquad \frac{e \xrightarrow{\mathsf{rd}(p_2,v_1,v_3)} e' \quad v_2:\tau_2 \in W(p_2) \quad v_3 = \mathsf{match}(v_1:\tau_1, v_2:\tau_2)}{W\,[p_1 \mapsto S[e:\tau]] \longrightarrow W\,[p_1 \mapsto S[e':\tau]]}$$

Table 3: The semantics at the concurrent level

can be either a '?' or a concrete value.

The types of patterns are all-important; here we shall assume the presence of the needed types (as provided by the type reconstruction algorithm of Section 4.4). The predicate match first matches the types of the values and then the values themselves using the auxiliary function vmatch. If successful the result of match is the value to be the result of the in or the rd.

$$\mathsf{match}(v_1 : \tau\text{"}pat\text{"}, v_2 : \tau) \quad = \quad \mathsf{vmatch}(v_1, v_2)$$

where vmatch is defined as

$$\begin{array}{lcl}
\mathsf{vmatch}(v, v) & = & v \text{ if } v \text{ is not a } \lambda\text{-abstraction} \\
\mathsf{vmatch}(?, v) & = & v \\
\mathsf{vmatch}(v, ?) & = & v \\
\mathsf{vmatch}(<\!\mathsf{pair}\ v_1\ v_2\!>, <\!\mathsf{pair}\ v_3\ v_4\!>) & = & <\!\mathsf{pair}\ \mathsf{vmatch}(v_1, v_3)\ \mathsf{vmatch}(v_2, v_4)\!> \\
\mathsf{vmatch}(<\!\mathsf{cons}\ v_1\ v_2\!>, <\!\mathsf{cons}\ v_3\ v_4\!>) & = & <\!\mathsf{cons}\ \mathsf{vmatch}(v_1, v_3)\ \mathsf{vmatch}(v_2, v_4)\!>
\end{array}$$

Note that λ-abstractions can only be matched by '?'.

It is possible to output values containing '?' into the process space. To input a '?' the pattern used must be a pattern for a pattern, i.e. have the type (τ pat) pat.

3 A small example

In this section we will demonstrate how to create parallel programs in λinda by creating a parallel quicksort algorithm. The main function in the quicksort algorithm is the function split which splits a list into two parts: one where the elements are smaller than or equal to the pivot element and one where the elements are larger. The function split looks like this:

```
split=λpivot·λlist·
  if isnil list then
    ([],[])
  else
    let
      element=head list
      result=split pivot (tail list)
    in
      if element¡=pivot then
        (element::(fst result),snd result)
      else
        (fst result,element::(snd result))
```

In a sequential quicksort the sort function is called recursively on the splitted parts of the list, after which the parts are concatenated into a single sorted list. A sequential version of quicksort could look like:

```
quicksort=λlist·
  let
    result=split (head list) list
    smaller=quicksort (fst result)
    larger=quicksort (snd result)
  in
    smaller++larger
```

Note that we use '++' as concatenation of lists and for convenience the pivot element is chosen as the first in the list.

The idea in the parallel version is to spawn a new process for each recursive function call. So each time a list is divided into two parts by split two new processes are started to sort the parts. When the new processes terminate they become passive elements in the process space containing the sorted sublists. Now the original process needs to collect the results in the right order. So we need to know how to find the right parts of the list which we want to combine into the resulting list. To this end each result of the partial result is tagged by a number indicating which part of the list it is, such that each time a split occurs the two new processes are numbered $2n$ and $2n+1$.

```
sorter=λps·λpart·λlist·
  if isnil list then
    (part,list)
  else
    let
      (smaller,larger)=split (head list) list
    in
      eval ps (λu·sorter ps (2*part) smaller);
      eval ps (λu·sorter ps (2*part+1) larger);
      let
        smaller=snd (in ps (2*part,?))
        larger=snd (in ps (2*part+1,?))
      in
        (part,smaller++larger)
```

Finally, we need the main function starting the quicksort:

```
quicksortpar=λlist·snd (sorter (makeps ()) 1 list)
```

This example shows a commonly used principle in λinda programs, which is to tag the values in the processes space such that it is possible to decide which value is the result of which process. In this example the tag is a unique number but often the tag is a string indicating what the value is. This also provides the ability to distinguish between results from different programs/algorithms evaluating in parallel.

4 An effect type system for λinda

In the rest of our paper we shall consider an effect type system for λinda. The ype of an expression is $\tau\&\kappa$, τ being the ordinary type and κ the *effect* of the expression, denoting the communication capabilities of the expression; inspired by work of Nielson and Nielson [NN93, NN94], effects are terms from a process calculus and can be given an operational semantics, giving us a tool to show that the behavioural effect of an expression is indeed sensible.

The types τ have the following syntax:

$$\tau \ ::= \ \mathcal{B} \mid \alpha \mid \tau_1 \times \tau_2 \mid \tau \ \mathsf{list} \mid \tau_1 \xrightarrow{\beta} \tau_2 \mid \tau \ \mathsf{pat} \mid \mathsf{ps}\ \rho$$

where $\alpha \in \mathbf{TypVar}$ denotes type variables.

The syntax of process behaviour types **Bexp** is:

$$\begin{array}{lcl} \beta & ::= & \beta+\beta \mid \beta;\beta \mid \mu b{\cdot}\beta \mid b \mid a \mid \varepsilon \\ a & ::= & \mathsf{out}(\rho,\tau) \mid \mathsf{eval}(\rho,\tau\&\beta) \mid \mathsf{rd}(\rho,\tau) \mid \mathsf{in}(\rho,\tau) \mid \mathsf{makeps}(\rho) \end{array}$$

Here +' denotes nondeterministic choice, ';' is sequencing, $\mu b{\cdot}\beta$ describes recursive behaviour, where b ranges over behaviour variables **BehVar**, $a \in$ **Act** is the possible actions, and ε denotes the empty behaviour. $\rho \in \mathbf{Reg}$ denotes regions.

Regions, though not essential to our approach, facilitate the semantics of behaviour types presented in Section 4.5. A region intuitively corresponds to the notion of process space – a region is a set of "program points" stating which process spaces a Linda primitive have access to. A region identifier should be thought of as being associated with occurrences of makeps. Regions ρ are given by:

$$\rho \quad ::= \quad i \mid r \mid \rho_1 \cup \rho_2$$

where $i \in \mathbf{RegId}$ denotes the region identifiers, $r \in \mathbf{RegVar}$ denotes region variables and '$\cup$' denotes the union of regions.

4.1 Determining the behaviour of a function

In [NN93] *subtyping* was used to give behaviour types to CML in a monomorphic setting where the programmer has to state the type of the arguments to functions; whereas *polymorphism* was employed in [NN94]. To see which solution is the most appropriate consider the following two problems.

The Argument Problem

The *argument problem* arises from the fact that functions bind behaviour. If the argument of a function is itself a function, the behaviour of the function may depend on the behaviour of the argument. A related problem is that of the latent behaviour of a function input from a process space. At compile time we have no simple way of determining this behaviour.

To see how the argument problem is handled using subtyping consider this expression

$$\lambda \mathsf{f}{\cdot}(\mathsf{f}\ (\mathsf{in\ ps}(?:\mathsf{int})) {+} 7$$

The type of this expression should be $(\mathsf{int} \xrightarrow{\beta'} \mathsf{int}) \xrightarrow{\beta} \mathsf{int}$ where the behaviour β of the function should contain the behaviour β' of f. Since we want an upper bound for the possible communication we have to assume that the argument has every possible behaviour. If we let ∞ denote every possible behaviour the functions get the type

$$(\mathsf{int} \xrightarrow{\infty} \mathsf{int}) \xrightarrow{\mathsf{in}\ (\rho,\mathsf{int});\infty} \mathsf{int}$$

In the case of higher-order functions, the application of an argument yields another function. This function also has a bound behaviour but now we can be more specific about it as we know the behaviour of the argument given. Therefore we have to keep track of the origins of the ∞'s in the effect when dealing with successive applications and instantiate them correspondingly. This amounts to using the ∞'s as *polymorphic effect variables*. By using polymorphism at the level of effects, we can easily solve the argument problem- – we use a generic behaviour variable representing the unknown behaviour contained in f. The type of our expression becomes

$$\forall b \cdot (\mathsf{int} \xrightarrow{b} \mathsf{int}) \xrightarrow{\mathsf{in}\ (\rho,\mathsf{int});b} \mathsf{int}$$

The Unification Problem

The *unification problem* arises when a function takes more than one argument with the same type but different effects. For instance cons is given the type:

$$\forall \alpha \cdot \alpha \xrightarrow{\varepsilon} \alpha\ \mathsf{list} \xrightarrow{\varepsilon} \alpha\ \mathsf{list}$$

If α is instantiated to a function type then we need to handle the bound behaviours. To demand that the functions in a list all have exactly the same effect would make some programs ill-typed that would be well-typed without effect types. Ignoring the effect will not work, for if we extract and apply a function from the list, we need to know its effect.

The unification problem is easily handled with subtyping. Consider the following two expressions:

$$\lambda \mathsf{w} \cdot\ \mathsf{out\ ps\ 7} : \alpha \xrightarrow{\mathsf{out}(\rho,\mathsf{int})} \mathsf{unit}$$
$$\lambda \mathsf{w} \cdot \mathsf{in\ ps}'(? : \mathsf{unit}) : \alpha \xrightarrow{\mathsf{in}\ (\rho',\mathsf{unit})} \mathsf{unit}$$

If these two functions should both occur in a list we need to give them the same type, i.e. find an upper bound for their behaviour:

$$\alpha \xrightarrow{\beta} \mathsf{unit} \qquad \text{where } \beta = \mathsf{out}(\rho, \mathsf{int}) + \mathsf{in}(\rho', \mathsf{unit})$$

The unification problem is not handled as elegantly using polymorphism. If we were to give the two expressions the same behaviour type using polymorphism we must extend each behaviour with a behaviour variable and unify them. This means that all functions including those with an empty behaviour need a behaviour variable. For example head gets the type

$$\forall\alpha\forall b{\cdot}\alpha\ \mathsf{list} \xrightarrow{\varepsilon+b} \alpha$$

We find this solution clumsy because the type will contain behaviour variables which often will not need to be instantiated. In the above example b might never be instantiated.

We are led to conclude that neither of the existing solutions will suffice in itself; subtyping solves the unification problem elegantly whereas polymorphism solves the argument problem elegantly. For λinda we combine the solutions using polymorphism to describe how the behaviour of a function argument is related to the behaviour of the function applied and using subtyping to handle arguments related by the same type variable.

4.2 Ordering Behaviour Types

Our subtype ordering expresses that one behaviour has fewer communication capabilities than another. As we saw above, we need a supremum operator $+$.

Definition 4.1
We define $(\mathbf{Bexp}, \leq)$ *to be the least preorder satisfying:*

[Supremum]	$\beta_1 \leq \beta_1 + \beta_2$	$\beta_2 \leq \beta_1 + \beta_2$
	$\beta + \beta \leq \beta$	
[Structural]	$\dfrac{\beta_1 \leq \beta_1' \qquad \beta_2 \leq \beta_2'}{\beta_1;\beta_2 \leq \beta_1';\beta_2'}$	$\dfrac{\beta_1 \leq \beta_1' \qquad \beta_2 \leq \beta_2'}{\beta_1 + \beta_2 \leq \beta_1' + \beta_2'}$
[Identity]	$\beta \leq \beta;\varepsilon$	$\beta;\varepsilon \leq \beta$
	$\beta \leq \varepsilon;\beta$	$\varepsilon;\beta \leq \beta$
[Unfold]	$\mu b{\cdot}\beta \leq \beta\{\mu b{\cdot}\beta/b\}$	$\beta\{\mu b{\cdot}\beta/b\} \leq \mu b{\cdot}\beta$

Moreover we need an ordering on regions:

Definition 4.2
We define $(\mathbf{Reg}, \leq)$ *as the least preorder for which* $\cup$ *acts as a supremum.*

Now we can define an ordering on types taking latent behaviours and regions into account.

Definition 4.3
We define $(\mathbf{Typ}, \leq)$ *as the least preorder satisfying*[1]*:*

[1] For simplicity we have excluded some rules needed to handle wildcards, see [KJH94]

$$
\begin{array}{rcll}
\tau & \leq & \tau\ \mathsf{pat} & \\
\tau_1\ \mathsf{list} & \leq & \tau_2\ \mathsf{list} & \text{if}\ \ \tau_1 \leq \tau_2 \\
\tau_1 \times \tau_2 & \leq & \tau_3 \times \tau_4 & \text{if}\ \ \tau_1 \leq \tau_3\ \wedge\ \tau_2 \leq \tau_4 \\
\tau_1 \xrightarrow{\beta_1} \tau_2 & \leq & \tau_3 \xrightarrow{\beta_2} \tau_4 & \text{if}\ \ \tau_3 \leq \tau_1\ \wedge\ \tau_2 \leq \tau_4\ \wedge\ \beta_1 \leq \beta_2 \\
\mathsf{ps}\ \rho_1 & \leq & \mathsf{ps}\ \rho_2 & \text{if}\ \ \rho_1 \leq \rho_2
\end{array}
$$

4.3 Typing Rules

Judgements are written $\Gamma \vdash e : \tau \& \beta$ and state that in type environment Γ the expression e has type τ and when evaluated the behaviour β. Since we have strict evaluation, variables and constants have no behaviour (except behaviour bound in a function type), so type environments have functionality $\Gamma : \mathbf{Var} \hookrightarrow \mathbf{TypSch}$. The type inference rules are given in Table 4.

[abs] specifies that abstraction captures the behaviour associated with the body. [app] reflects the call-by-value semantics; the effect of evaluating an application is the effect of evaluating the function to be applied, followed by the effect of evaluating the argument and finally the effect of the application itself. Conditionals are handled by [cond] – if the branches of a conditional have different behaviours, the subsumption rule is used to give them the same behaviour. [rec] is similar to the abstraction rule except that we have to extend the type environment with assumptions about the type of the recursion variable.

We let $t \in \mathbf{TypSch}$ denote type schemes:

$$t ::= \tau \mid \forall \alpha \cdot t \mid \forall b \cdot t \mid \forall r \cdot t$$

We write $\tau \prec t$ to state that the type τ is an instance of the type scheme t. The type schemes of the constants are given by the function TypeOf (Table 5).

The rule [let] makes use of the function Gen which generates a type scheme given a type environment and a type by quantifying over variables not bound in Γ:

$$
\begin{array}{rcl}
\mathsf{Gen}(\Gamma, \tau) & = & \mathsf{let}\ \{\alpha_1, \ldots, \alpha_n, b_1, \ldots, b_m, r_1, \ldots, r_l\} = \{\kappa \mid \kappa \in FV(\tau) \setminus FV(\Gamma)\} \\
& & \mathsf{in}\ \forall \alpha_1 \ldots \forall \alpha_n \forall b_1 \ldots \forall b_m \forall r_1 \ldots \forall r_l \cdot \tau
\end{array}
$$

where $\kappa \in \mathbf{TypVar} \cup \mathbf{BehVar} \cup \mathbf{RegVar}$.

The type system has the property that the type is preserved during evaluation.

Theorem 4.4 (Local Preservation)
If $\vdash e : \tau \& \beta$ *and* $e \xrightarrow{\varepsilon} e'$ *then* $\vdash e' : \tau \& \beta$.

$$\text{[const]} \qquad \Gamma \vdash c : \tau \& \varepsilon \quad \text{if } \tau \prec \mathsf{TypeOf}(c)$$

$$\text{[var]} \qquad \Gamma \vdash x : \tau \& \varepsilon \quad \text{if } \tau \prec \Gamma(x)$$

$$\text{[sub]} \qquad \frac{\Gamma \vdash e : \tau_1 \& \beta_1}{\Gamma \vdash e : \tau_2 \& \beta_2} \qquad \text{if } \tau_1 \leq \tau_2 \text{ and } \beta_1 \leq \beta_2$$

$$\text{[add-hyp]} \qquad \frac{\Gamma \vdash e : \tau \& \beta}{\Gamma[x \mapsto t] \vdash e : \tau \& \beta} \qquad \text{if } x \notin *FV*(e)$$

$$\text{[abs]} \qquad \frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2 \& \beta}{\Gamma \vdash \lambda x {\cdot} e : \tau_1 \xrightarrow{\beta} \tau_2 \& \varepsilon}$$

$$\text{[app]} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \xrightarrow{\beta_3} \tau_1 \& \beta_1 \qquad \Gamma \vdash e_2 : \tau_2 \& \beta_2}{\Gamma \vdash e_1\, e_2 : \tau_1 \& \beta_1 ; \beta_2 ; \beta_3}$$

$$\text{[cond]} \qquad \frac{\Gamma \vdash e : \mathtt{bool} \& \beta' \quad \Gamma \vdash e_1 : \tau \& \beta \quad \Gamma \vdash e_2 : \tau \& \beta}{\Gamma \vdash \ \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau \& \beta' ; \beta}$$

$$\text{[let]} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \& \beta_1 \quad \Gamma[x \mapsto \mathsf{Gen}(\Gamma, \tau_1)] \vdash e_2 : \tau_2 \& \beta_2}{\Gamma \vdash \ \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \& \beta_1 ; \beta_2}$$

$$\text{[rec]} \qquad \frac{\Gamma[x \mapsto \tau_1 \xrightarrow{\beta} \tau_2, y \mapsto \tau_1] \vdash e : \tau_2 \& \beta}{\Gamma \vdash \mathsf{rec}\, x(y) {\cdot} e : \tau_1 \xrightarrow{\beta} \tau_2 \& \varepsilon}$$

$$\text{[clos]} \qquad \frac{\Gamma \vdash c : \tau_1 \xrightarrow{\varepsilon} \cdots \xrightarrow{\varepsilon} \tau_n \& \varepsilon \quad \Gamma \vdash v_i : \tau_i \& \varepsilon}{\Gamma \vdash {<} c\, v_1 \cdots v_m {>} : \tau_{m+1} \xrightarrow{\varepsilon} \cdots \xrightarrow{\varepsilon} \tau_n \& \varepsilon} \qquad \text{for } 1 \leq i \leq m < n$$

Table 4: Type inference rules.

c	TypeOf
pair	$\forall\alpha\forall\alpha'\cdot\alpha \xrightarrow{\varepsilon} \alpha' \xrightarrow{\varepsilon} \alpha\times\alpha'$
cons	$\forall\alpha\cdot\alpha \xrightarrow{\varepsilon} \alpha\ \mathsf{list} \xrightarrow{\varepsilon} \alpha\ \mathsf{list}$
in	$\forall r\forall\alpha\cdot\mathsf{ps}\ r \xrightarrow{\varepsilon} \alpha\,\mathsf{pat} \xrightarrow{\mathsf{in}(r,\alpha)} \alpha$
rd	$\forall r\forall\alpha\cdot\mathsf{ps}\ r \xrightarrow{\varepsilon} \alpha\,\mathsf{pat} \xrightarrow{\mathsf{rd}(r,\alpha)} \alpha$
out	$\forall r\forall\alpha\cdot\mathsf{ps}\ r \xrightarrow{\varepsilon} \alpha \xrightarrow{\mathsf{out}(r,\alpha)} \mathtt{unit}$
eval	$\forall r\forall\alpha\forall b\cdot\mathsf{ps}\ r \xrightarrow{\varepsilon} (\mathsf{unit} \xrightarrow{b} \alpha) \xrightarrow{\mathsf{eval}(r,\alpha\&b)} \mathsf{unit}$
makeps	$\forall r\cdot\mathsf{unit} \xrightarrow{\mathsf{makeps}(r)} \mathsf{ps}\ r$

Table 5: The types of a sample of the constants. In makeps we will only instantiate r to region identifiers.

4.4 A type reconstruction algorithm

Our type reconstruction algorithm $\mathcal{T}$ extends the standards ideas of [Mil78] – we get a type-behaviour pair using the behaviour reconstruction algorithm $\mathcal{W}_\beta$:

$$\mathcal{T}(e) = (\tau,\beta)\ \text{ where }\ (\Theta,\tau,\beta) = \mathcal{W}_\beta(\mathsf{nil}, e)$$

Here nil is the empty type environment and Θ is a substitution on type and behaviour variables found by an auxiliary unification algorithm $\mathcal{U}$.

As we have seen, the main problem in inferring behaviour types arises from the handling of higher-order functions. Consider e.g. a list of functions cons (rd ps) (cons (ps)nil); all the functions should have the same type τ but not necessarily the same effect. Naïvely we want the list to have the type $(\alpha\ \mathsf{pat} \xrightarrow{\mathsf{rd}(\rho,\alpha)+\mathsf{in}(\rho,\alpha)} \alpha)\ \mathsf{list}$.

To see how this can be obtained consider the first part cons (rd ps). This must be given the type $(\alpha\ \mathsf{pat} \xrightarrow{b} \alpha)\ \mathsf{list} \xrightarrow{\varepsilon} (\alpha\ \mathsf{pat} \xrightarrow{\mathsf{rd}(\rho,\alpha)+b} \alpha)\ \mathsf{list}$.

In a traditional type system cons has the type $\alpha \to \alpha\ \mathsf{list} \to \alpha\ \mathsf{list}$. This is not adequate here, as the α's are not instantiated to the same type. We therefore parameterize the α's with behaviour variables:

$$\mathsf{TypeOf}(\mathsf{cons}) \;=\; \forall\alpha\forall b\forall b'\cdot\alpha(b) \xrightarrow{\varepsilon} \alpha(b')\ \mathsf{list} \xrightarrow{\varepsilon} \alpha(b+b')\ \mathsf{list}$$

This approach however, will not yield the expected type but rather $(\alpha\ \mathsf{pat} \xrightarrow{\mathsf{rd}(\rho,\alpha)+\mathsf{in}(\rho,\alpha)+b} \alpha)\ \mathsf{list}$ as we have to assign some behaviour to nil.

In the presence of parameterized type variables, simple substitutions are not sufficient as a means of instantiation and the unification algorithm $\mathcal{U}$ must be redefined – only in the case of application should behaviour variables be instantiated whereas behaviours should be summed in other cases. The full details of $\mathcal{U}$ are described in [KJH94].

The type reconstruction algorithm is *sound*:

Theorem 4.5 *If* $\Gamma \vdash e$ *is* closed *and* $(\Theta, \tau, \beta) = \mathcal{W}_\beta(\Gamma, e)$ *then* $\Theta\Gamma \vdash e : \tau\&\beta$

4.5 Semantics of Bexp

We can also give an operational semantics for the behaviour expressions. The semantics is given by a labelled transition system with labels:

$$m \in \mathbf{Act} \cup \{\varepsilon\}$$

We add a new element '$\heartsuit$' to the set of behaviours to indicate termination (no λinda expression has this behaviour.)

The semantics again consists of sequential and a concurrent level. The rules at the sequential level are shown in Table 6. Note that as the choice in '+' does not depend on the communication capabilities of a behaviour, '+' is the *internal choice* of [Hen88] rather than the '+' of CCS [Mil89].

[act]	$a \xrightarrow{a} \varepsilon$	$[\text{seq}]_1$	$\dfrac{\beta_1 \xrightarrow{m} \beta_1'}{\beta_1; \beta_2 \xrightarrow{m} \beta_1'; \beta_2}$
[idle]	$\beta \xrightarrow{\varepsilon} \beta$		
[term]	$\varepsilon \xrightarrow{\varepsilon} \heartsuit$	$[\text{seq}]_2$	$\dfrac{\beta \xrightarrow{m} \heartsuit}{\beta; \beta' \xrightarrow{m} \beta'}$
$[\text{choice}]_1$	$\beta_1 + \beta_2 \xrightarrow{\varepsilon} \beta_1$		
$[\text{choice}]_2$	$\beta_1 + \beta_2 \xrightarrow{\varepsilon} \beta_2$	[rec]	$\mu b{\cdot}\beta \xrightarrow{\varepsilon} \beta\{\mu b{\cdot}\beta / b\}$

Table 6: Sequential semantics for behaviour types.

For each element in a process space we have a configuration (β, τ, ρ) and the concurrent level is modelled as a multiset of such configurations. In Table 7 the rules for the concurrent part of the semantics are shown. [PS-elem] states that any element in a process space may evolve separately and [PS-uni] states that any part of the multiset may evolve. The rest of the rules state the usual semantics for the Linda primitives. Note that in the rules [PS-in] and [PS-rd] the regions must have common region identifiers in order to reflect a possible in or rd.

The behaviour type of an expression can be seen as an abstract description of its possible evaluation. We define a relation between labels of the labelled transition system for expressions and labels of the transition system for behaviour expressions:

$$[\text{PS-elem}] \quad \frac{\beta \xrightarrow{\varepsilon} \beta'}{\{\!\![(\beta,\tau,\rho)]\!\!\} \to \{\!\![(\beta',\tau,\rho)]\!\!\}}$$

$$[\text{PS-in}] \quad \frac{\beta \xrightarrow{\mathsf{in}(\rho'',\tau')} \beta'}{\{\!\![(\beta,\tau,\rho),(\heartsuit,\tau',\rho')]\!\!\} \to \{\!\![(\beta',\tau,\rho)]\!\!\}} \quad \rho'\cap\rho''\neq\emptyset$$

$$[\text{PS-rd} \quad \frac{\beta \xrightarrow{\mathsf{rd}(\rho'',\tau')} \beta'}{\{\!\![(\beta,\tau,\rho),(\heartsuit,\tau',\rho')]\!\!\} \to \{\!\![(\beta',\tau,\rho),(\heartsuit,\tau',\rho')]\!\!\}} \quad \rho'\cap\rho''\neq\emptyset$$

$$[\text{PS-out}] \quad \frac{\beta \xrightarrow{\mathsf{out}(\rho',\tau')} \beta'}{\{\!\![(\beta,\tau,\rho)]\!\!\} \to \{\!\![(\beta',\tau,\rho),(\heartsuit,\tau',\rho')]\!\!\}}$$

$$[\text{PS-eval}] \quad \frac{\beta \xrightarrow{\mathsf{eval}(\rho',\tau'\&\beta')} \beta''}{\{\!\![(\beta,\tau,\rho)]\!\!\} \to \{\!\![(\beta'',\tau,\rho),(\beta',\tau',\rho')]\!\!\}}$$

$$[\text{PS-uni}] \quad \frac{PS_1 \ \to PS_1'}{PS_1 \uplus PS_2 \to PS_1' \uplus PS_2}$$

Table 7: Concurrent evaluation of behaviour types.

Definition 4.6 *We define the relation $\asymp$ by the following rules:*

$$\begin{array}{lcll}
\mathsf{in}(v_1,v_2,v_3) & \asymp & \mathsf{in}(\rho,\tau) & \textit{if } \vdash v_1 : \mathsf{ps}\ \rho\&\varepsilon \textit{ and } \vdash v_3 : \tau\&\varepsilon \\
\mathsf{rd}(v_1,v_2,v_3) & \asymp & \mathsf{rd}(\rho,\tau) & \textit{if } \vdash v_1 : \mathsf{ps}\ \rho\&\varepsilon \textit{ and } \vdash v_3 : \tau\&\varepsilon \\
\mathsf{out}(v_1,v_2) & \asymp & \mathsf{out}(\rho,\tau) & \textit{if } \vdash v_1 : \mathsf{ps}\ \rho\&\varepsilon \textit{ and } \vdash v_2 : \tau\&\varepsilon \\
\mathsf{eval}(v,e) & \asymp & \mathsf{eval}(\rho,\tau\&\beta) & \textit{if } \vdash v : \mathsf{ps}\ \rho\&\varepsilon \textit{ and } \vdash e : \tau\&\beta \\
\mathsf{makeps}(v) & \asymp & \mathsf{makeps}(\rho) & \textit{if } \vdash v : \mathsf{ps}\ \rho\&\varepsilon \\
\varepsilon & \asymp & \varepsilon &
\end{array}$$

Using this relation the following theorem states that the semantics of behaviours indeed mimics the semantics of expressions: The type of an expression does not change but the effect does.

Theorem 4.7
If $\vdash e : \tau\&\beta$ and $e \xrightarrow{a} e'$ then there exist β' and a' such that $\beta \xrightarrow{a'} \beta'$, $a \asymp a'$ and $\vdash e' : \tau\&\beta'$.

As an immediate corollary of Theorem 4.7 we have that when λinda expressions evaluate, their associated behaviours decrease.

Corollary 4.8
If $\vdash e : \tau\&\beta$ *and* $e \xrightarrow{l} e'$ *and* $\vdash e' : \tau\&\beta'$ *then there exists* $m \asymp l$ *such that* $m; \beta' \leq \beta$.

An important difference from the behaviour semantics in [NN93] is that we do not use the ordering of behaviours in the semantics. Instead, the ordering can be recovered from our semantics by a simulation ordering in the sense of [Mil89]. We use $\beta \stackrel{m}{\Longrightarrow} \beta'$ to specify that β can evaluate to β' using exactly one m-transition and zero or more ε-transitions, i.e. $\stackrel{m}{\Longrightarrow} = \xrightarrow{\varepsilon} * \xrightarrow{m} \xrightarrow{\varepsilon} *$.

Definition 4.9
A binary relation $S \subseteq \mathbf{Bexp} \times \mathbf{Bexp}$ *is a simulation if* $(\beta_1, \beta_2) \in S$ *implies that if*

$$\beta_1 \xrightarrow{m} \beta_1'$$

then there exists a β_2' *such that:*

$$\beta_2 \stackrel{m}{\Longrightarrow} \beta_2' \qquad \textit{and} \qquad (\beta_1', \beta_2') \in S$$

We write $\beta \preceq \beta'$ *if* (β, β') *is contained in some simulation.*

Theorem 4.10 (Soundness)
If $\beta \leq \beta'$ *then* $\beta \preceq \beta'$.

5 Conclusions and further work

In this paper we present a functional language based on the Linda concept and an effect type system for describing concurrent behaviour. Behaviour types are rich in information about the causality of Linda primitives and which expressions are input and output to the process spaces. The semantics of behaviours shows that the behaviour type of an expression indeed describes its concurrent aspects. Further we have shown that the ordering on behaviours is sound with respect to the given simulation. An open problem at the time of writing is whether the type reconstruction algorithm for behaviour types is complete. We conjecture that this indeed the case. Further, it should be examined whether our type inference rules have the principal type property.

Combining subtyping and polymorphism in order to solve the argument and unification problems elegantly gives the programmer the most intuitive types (although it complicates the type inference algorithm for the behaviour types somewhat). The argument and unification problems also occur in other concurrent functional languages such as CML, Facile and LCS, and the ideas from the type inference algorithm for behaviour types presented here could prove fruitful in these languages.

In [KJH94] we give another, sort-based effect type system. Compared to behaviours, sorts are less descriptive and do not capture the causality of the Linda primitives but only state which expressions are input and output. As there is a straightforward and effective translation of behaviour types to sorts which preserves welltypedness, the combination of this with our type reconstruction algorithm for behaviour effect types provides us with a sort inference algorithm.

Several extensions of the type system presented here can be imagined. In particular, one should consider the possibility of *explicit polymorphism* as this would allow us to communicate polymorphic values directly.

References

[Ber93] Bernard Berthomieu. Programming with Behaviors in an ML framework. The Syntax and Semantics of LCS, April 1993.

[BD94] Dominique Bolignano and Mourad Debabi A Semantic Theory for Concurrent ML *Proceedings of Theoretical Aspects of Computer Software (TACS '94)*, pages 766–785, Springer LNCS 789.

[GB82] David Gelernter and Arthur J. Bernstein. Distributed Communication via Global Buffer. *ACM Symposium on Principles of Distributed Computing*, pages 10–18, August 1982.

[GMP89] A. Giacalone, P. Mishra, and S. Prasad. Facile: A Symmetric Integration of Concurrent and Functional Programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

[Hen88] Matthew Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[KJH94] Josva Kleist, Bo Jensen, and Martin Hansen. Effect Type Systems for Concurrent Functional Languages with Linda Primitives, January 1994. Student Report — Aalborg University.

[Luc87] J.M. Lucassen. Type and Effects: Towards an Integration of Functional and Imperative Programming, 1987. PhD thesis, Laboratory of Computer Science, MIT.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MTH90] Robin Milner, Mads Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[NN93] Flemming Nielson and Hanne Riis Nielson. From CML to Process Algebras. Technical report, Computer Science Department - Aarhus University, March 1993.

[NN94] Hanne Riis Nielson and Flemming Nielson. Higher-Order Concurrent Programs with Finite Communication Topology. To appear in ACM SIGPLAN-SIGACT, 1994.

[Rep92] John H. Reppy. *Higher-Order Concurrency.* PhD thesis, Department of Computer Science, Cornell University, June 1992.

[Rey89] John C. Reynolds. Syntactic Control of Interference — Part 2. *Automata Languages and Programming, 16th International Colloquium (ICALP '89)*, pages 704–722, Springer LNCS 372.

[Tho93] Bent Thomsen. Polymorphic Sort and Types for Concurrent Functional Programs. Technical report, European Computer-Industry Research Centre, June 93. ECRC-93-10.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. The Type and Effect Discipline. In Anne Copeland, editor, *Proceedings - Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 162–173. IEEE, IEEE Computer Society Press, 1992.