Constraints for Polymorphic Behaviours of Concurrent ML

Flemming Nielson, Hanne Riis Nielson

DAIMI, Computer Science Department,
Aarhus University (Bldg. 540), Ny Munkegade,
DK-8000 Aarhus C, Denmark.

To appear in Proceedings CCL'94 (Springer Lecture Notes in Computer Science).

We present a type and behaviour reconstruction algorithm for Standard ML with concurrency. The behaviours express the communication effects during execution and resemble terms of a process algebra. The algorithm uses unification for the (essentially) free algebra of types and algebraic reconstruction for collecting constraints for the non-free algebra of behaviours. The algorithm and the statement and proof of soundness are designed so as to make no assumptions on the existence of "principal" behaviours as these are unlikely to exist. The main complication is that the notion of expansiveness does not suffice for a sufficiently general treatment of the polymorphic `let`-construct.

1 Introduction

Currently there is a growing interest in so-called multiparadigmatic languages that attempt to obtain the best of two worlds by integrating the features of more than one programming paradigm. In this paper we study CML [11] that extends Standard ML with primitives for communication; other languages with similar aims include Facile[2], PolyML[7] and LCS [1]. The efficient implementation of such languages is a major task and places new demands on the technology for program analysis so as to provide correct and useful information about the behaviour of programs. Furthermore program analysis may be used to provide information that may help programmers to maintain important invariants by extracting information that may otherwise be deeply buried in the code. An example is the link between multiparadigmatic languages of the CML-variety and process algebras of the CCS-variety first pointed out in [9]. Here program analysis is used to extract the communication behaviour of CML-programs in the form of a process algebra: which communications take place, in what order, over which channels, and what are the types of entities communicated. As demonstrated in [10] this can be used to obtain a clearer understanding of the behaviour of programs which leads to more resource-conscious implementations that allow to dispense with multi-tasking and multi-plexing for well-behaved programs.

The advantages of strong typing are well-known as exemplified by the growing success of Standard ML and similar languages. An increasingly popular trend in program analysis is to express additional intensional program properties in notations reminiscent of type systems: we shall use the term annotated type systems for such notations. An early approach was the use of strictness types to analyse lazy (call-by-name) functional languages for which function parameters were surely needed and hence could be evaluated before entering the function (e.g. [5]). Another noteworthy development is the use of effect systems to analyse side-effects, aspects of communication behaviour, and the possibility of parallel execution on vector processors. Our use of communication behaviours falls within this approach but retains much more causal information in the communication behaviours.

The specification of program analyses thus takes the form of presenting an inference system for annotated types that generalises the more common inference systems for ordinary types, an example of which is the Damas-Milner inference system for Standard ML. The methods distinguish themselves from more general logical methods, e.g. the use of strictness logics [3], by the demand to maintain decidability of the inference system: the hope is that modifications of algorithms for type inference, an example being Milner's algorithm $\mathcal{W}$, may suffice for obtaining also the program analysis information of interest. A noteworthy paper along these lines is [4] that develops an algebraic reconstruction algorithm for a simple effect system for the higher-order polymorphic language KFX. Robinson unification is used for the type structure, which is a free algebra, whereas constraint collection is used for the effect structure, which is a non-free algebra; these constraints then have to be solved and in the case of [4] there is one associative and one commutative law so that one may use UCAI-unification (see [14] for references).

A main limitation of most current algorithms for extracting effect information (including [4]), and similarly for much work on admitting subtypes (e.g. [8]), is the proper treatment of the polymorphic `let`-construct of Standard ML. There are two reasons for this: one is that the presence of side-effects (e.g. in the form of communications) makes it impossible to expand the `let`-construct by syntactically substituting the `let`-bound expression into the body of the `let`; the other is that we shall want to exploit the effect information to obtain more general solutions than provided by the rather simple-minded distinction between expansive and non-expansive `let`-constructs. To obtain a general treatment of the `let`-construct in the case of side-effects, [12] departs from [4] by modifying the notion of algebraic reconstruction and avoiding the explicit use of unification in non-free algebras. In particular, constraints take the form of inclusions (in our notation $d \leq \beta$) rather than equations (as in $d = \beta$), a special canonical solution to a set of constraints is defined, and this canonical solution is used extensively during the unification steps of the algorithm. A key result shows that the canonical solution is principal in the sense of Robinson unification in that it provides the most general substitution (under composition of substitutions).

Since the extraction of communication behaviours for CML involves a general treatment of the `let`-construct it is natural to base our algorithm on [12] as opposed to [4]. However, our non-free algebra of effects (which we call behaviours) has much more algebraic structure than [12]. Although a notion of canonical solution can still be defined we have been unable to find a suitable notion of principality. Consequently our algorithm will build on ingredients from both [12] and [4] but many aspects need to be carefully revised (not least concerning which type variables to generalise in the `let`-construct), new notions developed, and special attention paid to generic (or polymorphic) variables in the formulation of soundness. While our algorithm is specifically developed for the extraction of behaviours for CML [10] we believe the ideas to be useful in general when principality of solutions to constraints cannot be ascertained, an example being extending [4] to deal properly with the polymorphic `let`-construct.

In summary we believe that this paper presents the first type and behaviour/effect reconstruction algorithm that (1) gives a general treatment of `let`-polymorphism and at the same time (2) allows the behaviours/effects to have complicated algebraic structure. The constraints collected by the algorithm need not have principal solutions so special care is taken to ensure that "incomparable" solutions cannot accidentally be mixed.

2 Type and Behaviour Inference

We consider a polymorphic subset of CML [11] where expressions $e \in \mathbf{Exp}$ are given by

$$\begin{array}{lll} e & ::= & c \mid x \mid \texttt{fn}\ x\ \texttt{=>}\ e \mid e_1\ e_2 \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \mid \texttt{rec}\ f\ x\ \texttt{=>}\ e \\ & \mid & \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \end{array}$$

Here x and f are program identifiers. In addition to function abstraction and function application we have a polymorphic `let`-construct, recursion and a conditional. The constants $c \in \mathbf{Const}$ are given by

$$\begin{array}{lll} c & ::= & \texttt{()} \mid \texttt{true} \mid \texttt{false} \mid n \mid \texttt{+} \mid \texttt{*} \mid \texttt{=} \mid \cdots \\ & \mid & \texttt{pair} \mid \texttt{fst} \mid \texttt{snd} \mid \texttt{nil} \mid \texttt{cons} \mid \texttt{hd} \mid \texttt{tl} \mid \texttt{isnil} \\ & \mid & \texttt{send} \mid \texttt{receive} \mid \texttt{choose} \mid \texttt{wrap} \mid \texttt{sync} \mid \texttt{channel} \mid \texttt{fork} \end{array}$$

We have constants corresponding to the base types `unit`, `bool` and `int` together with operations for constructing and destructing pairs and lists as well as the usual arithmetic and boolean operations. We shall allow to use a bit of syntactic sugar: so we write `[`e_1`,`e_2`]` for `cons` e_1 `(cons` e_2 `nil)`, we write `(`e_1`,`e_2`)` for `pair` e_1 e_2, and we write e_1`;`e_2 for e.g. `snd(`e_1`,`e_2`)`. The primitives of CML are available as operations for sending and receiving values over channels, for choosing between various communication possibilities and for modifying values being communicated. To be more precise these primitives only construct suspended communications that may be enacted using synchronisation. Finally, there are primitives for creating new channels and processes.

As usual we shall use *types* to classify the values that expressions can evaluate to. When executing a CML program channels and processes may be created and values may be communicated and we shall extend the type system with communication *behaviours* to record this. To be able to distinguish between the various channels, e.g. based on the program points where they are created, we introduce a notion of *regions* into the type system but the details will not be of importance for the developments of this paper.

For types $t \in \mathbf{Typ}$ we take

$$t \quad ::= \quad \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \mid \alpha \mid t_1 \to^b t_2 \mid t_1 \times t_2 \mid t\ \texttt{list} \mid t\ \texttt{chan}\ r \mid t\ \texttt{com}\ b$$

where α is a meta-variable for type variables (α, α_1, α' etc.). The function type is written $t_1 \to^b t_2$ indicating that the argument type of the functions is t_1, the result type is t_2 and the latent behaviour is b; this means that when a function is supplied with its argument the resulting behaviour of executing the function call will be b. The type of a channel is t `chan` r indicating that the channel is allocated in region r and that values of type t can be communicated over it. Finally, t `com` b is the type of a *suspended* communication: when it eventually is enacted using `sync`, it will result in a value of type t and the resulting behaviour will be b.

Formally, behaviours $b \in \mathbf{Beh}$ are given by

$$b \quad ::= \quad \epsilon \mid r!t \mid r?t \mid t \text{ CHAN } r \mid \beta \mid \text{FORK } b \mid b_1; b_2 \mid b_1 + b_2 \mid \text{REC } \beta. b$$

Here ϵ stands for the non-observable behaviour. We write $r!t$ for sending a value of type t over a channel in region r and similarly $r?t$ for receiving a value of type t over a channel in region r. The allocation of a new channel in region r is written t CHAN r where t is the type of values to be communicated. The behaviour FORK b expresses that a process with behaviour b is spawned. Behaviours may be combined using sequencing and choice and they may be recursive. We write β for a meta-variable for behaviour variables (β, β_1, β' etc.). So for example REC β. (t CHAN r + (FORK($r?t$); β)) is the behaviour of a program that either will create a channel and then no more communications take place, or it will spawn a process that inputs on some channel and then the overall process will recurse.

Finally, regions $r \in \mathbf{Reg}$ are given by

$$r \quad ::= \quad \mathrm{r}_0 \mid \mathrm{r}_1 \mid \cdots \mid \rho$$

Here ρ denotes a meta-variable for region variables (ρ, ρ_1, ρ' etc.) and r_0, r_1, ... denote region constants (which it may be instructive to think of as program points).

The *type schemes* are obtained from types by quantifying over type variables, behaviour variables and region variables: they have the form $\forall\vec{\alpha}\vec{\beta}\vec{\rho}.t$ where $\vec{\alpha}$, $\vec{\beta}$ and $\vec{\rho}$ are lists of variables. We shall occasionally allow to use γ as a meta-variable that ranges over α's, β's and ρ's as appropriate. A type t is a *generic instance* of a type scheme $ts = \forall\vec{\alpha}\vec{\beta}\vec{\rho}.t_0$, written $ts \succ t$ (or $ts \succ t$ under φ), if there exists a substitution φ such that $\varphi t_0 = t$ and the domain of φ is a subset of $\{\vec{\alpha}\vec{\beta}\vec{\rho}\}$. Here a *substitution* φ is a total function from type variables to types, from behaviour variables to behaviours, and from region variables to regions, such that all but a finite set of variables are mapped to themselves. The domain of the substitution φ is the finite set of variables not mapped to themselves and we write $\mathrm{Dom}(\varphi)$ for this set. Furthermore, a type scheme ts' is an instance of ts, written $ts \succeq ts'$, if whenever $ts' \succ t$ also $ts \succ t$.

Occasionally, the context may demand that a subexpression is given a type with a latent behaviour that is larger than what seems to be desired at a first glance. For an example consider the program

```
choose [send (ch1, 7), receive ch2]
```

The first element of the list has type `int com r1!int` (assuming `ch1` has type `int chan r1`) and the second element has type `int com r2?bool` (assuming `ch2` has type `int chan r2`). We want the elements of the list to have type `int com (r1!int + r2?int)` to record that either one of the branches may be chosen at run-time. So we need to coerce the types `int com r1!int` and `int com r2?int` into `int com (r1!int + r2?int)`.

As discussed in [10] there are several ways of achieving this. The decision made there is to adopt the approach of *early subsumption* where generic instantiations produce the required specialised types and we do not have subsumption rules for modifying the

c	TypeOf(c)
`+`	$\forall\beta_1,\beta_2.\ \texttt{int} \rightarrow^{\epsilon+\beta_1} \texttt{int} \rightarrow^{\epsilon+\beta_2} \texttt{int}$
`pair`	$\forall\alpha_1,\alpha_2,\beta_1,\beta_2.\ \alpha_1 \rightarrow^{\epsilon+\beta_1} \alpha_2 \rightarrow^{\epsilon+\beta_2} \alpha_1\times\alpha_2$
`fst`	$\forall\alpha_1,\alpha_2,\beta.\ \alpha_1\times\alpha_2 \rightarrow^{\epsilon+\beta} \alpha_1$
`snd`	$\forall\alpha_1,\alpha_2,\beta.\ \alpha_1\times\alpha_2 \rightarrow^{\epsilon+\beta} \alpha_2$
`send`	$\forall\alpha,\beta_1,\beta_2,\rho.\ (\alpha\ \texttt{chan}\ \rho)\times\alpha \rightarrow^{\epsilon+\beta_1} \alpha\ \texttt{com}\ (\rho!\alpha+\beta_2)$
`receive`	$\forall\alpha,\beta_1,\beta_2,\rho.\ (\alpha\ \texttt{chan}\ \rho) \rightarrow^{\epsilon+\beta_1} \alpha\ \texttt{com}\ (\rho?\alpha+\beta_2)$
`choose`	$\forall\alpha,\beta_1,\beta_2,\beta_3.\ (\alpha\ \texttt{com}\ \beta_1)\ \texttt{list} \rightarrow^{\epsilon+\beta_2} \alpha\ \texttt{com}\ (\beta_1+\beta_3)$
`wrap`	$\forall\alpha_1,\alpha_2,\beta_1,\beta_2,\beta_3,\beta_4.\ (\alpha_1\ \texttt{com}\beta_1)\times(\alpha_1 \rightarrow^{\beta_2} \alpha_2) \rightarrow^{\epsilon+\beta_3}$ $\alpha_2\ \texttt{com}\ ((\beta_1;\beta_2)+\beta_4)$
`sync`	$\forall\alpha,\beta_1,\beta_2.\ (\alpha\ \texttt{com}\ \beta_1) \rightarrow^{\beta_1+\beta_2} \alpha$
`channel`	$\forall\alpha,\beta,\rho.\ \texttt{unit} \rightarrow^{(\alpha\ \mathrm{CHAN}\ \rho)+\beta} (\alpha\ \texttt{chan}\ \rho)$
`fork`	$\forall\alpha,\beta_1,\beta_2.\ (\texttt{unit} \rightarrow^{\beta_1} \alpha) \rightarrow^{(\mathrm{FORK}\ \beta_1)+\beta_2} \texttt{unit}$

Table 1: Type Schemes for Selected Constants

$$tenv \vdash c : t\ \&\ b \text{ if TypeOf}(c) \succ t \text{ and } \epsilon \sqsubseteq b$$

$$tenv \vdash x : t\ \&\ b \text{ if } tenv(x) \succ t \text{ and } \epsilon \sqsubseteq b$$

$$\frac{tenv[x \mapsto t] \vdash e : t'\ \&\ b}{tenv \vdash \texttt{fn}\ x\ \texttt{=>}\ e : t \rightarrow^{b} t'\ \&\ b'} \text{ if } \epsilon \sqsubseteq b'$$

$$\frac{tenv \vdash e_1 : t \rightarrow^{b} t'\ \&\ b_1 \qquad tenv \vdash e_2 : t\ \&\ b_2}{tenv \vdash e_1\ e_2 : t'\ \&\ b'} \text{ if } b_1;b_2;b \sqsubseteq b'$$

$$\frac{tenv \vdash e_1 : t_1\ \&\ b_1 \qquad tenv[x \mapsto ts] \vdash e_2 : t_2\ \&\ b_2}{tenv \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : t_2\ \&\ b'} \text{ if } ts = \text{gen}(tenv, b_1)t_1 \text{ and } b_1;b_2 \sqsubseteq b'$$

$$\frac{tenv[f \mapsto t \rightarrow^{b} t'][x \mapsto t] \vdash e : t'\ \&\ b}{tenv \vdash \texttt{rec}\ f(x)\texttt{=>}e : t \rightarrow^{b} t'\ \&\ b'} \text{ if } \epsilon \sqsubseteq b'$$

$$\frac{tenv \vdash e : \texttt{bool}\ \&\ b \qquad tenv \vdash e_1 : t\ \&\ b_1 \qquad tenv \vdash e_2 : t\ \&\ b_2}{tenv \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t\ \&\ b'} \text{ if } b;(b_1+b_2) \sqsubseteq b'$$

Table 2: Type and Behaviour Inference

behaviours that label type constructors. A similar choice is made in [12]. This means that the latent behaviour of functions and suspended communications must always be prepared to be larger than what seems desired at a first glance. Hence whenever the behaviour b seems called for we shall write $b+\beta$ for a suitable behaviour variable β. This explains the type schemes of the selected constants given in Table 1. Note that only the constants `sync`, `channel` and `fork` have a non-trivial latent behaviour (i.e. have $b \neq \epsilon$).

• pre-order laws	**P1.** $b \sqsubseteq b$
	P2. if $b_1 \sqsubseteq b_2$ and $b_2 \sqsubseteq b_3$ then $b_1 \sqsubseteq b_3$
• pre-congruence laws	**C1.** if $b_1 \sqsubseteq b_2$ and $b_3 \sqsubseteq b_4$ then $b_1; b_3 \sqsubseteq b_2; b_4$
	C2. if $b_1 \sqsubseteq b_2$ and $b_3 \sqsubseteq b_4$ then $b_1 + b_3 \sqsubseteq b_2 + b_4$
	C3. if $b_1 \sqsubseteq b_2$ then $\textsc{fork}\ b_1 \sqsubseteq \textsc{fork}\ b_2$
	C4. if $b_1 \sqsubseteq b_2$ then $\textsc{rec}\ \beta.\ b_1 \sqsubseteq \textsc{rec}\ \beta.\ b_2$
• laws for sequencing	**S1.** $b_1; (b_2; b_3) \equiv (b_1; b_2); b_3$
	S2. $(b_1 + b_2); b_3 \equiv (b_1; b_3) + (b_2; b_3)$
• laws for ϵ	**E1.** $b \equiv \epsilon; b$
	E2. $b; \epsilon \equiv b$
• laws for choice (or join)	**J1.** $b_1 \sqsubseteq b_1 + b_2$ and $b_2 \sqsubseteq b_1 + b_2$
	J2. $b + b \sqsubseteq b$
• laws for recursion	**R1.** $\textsc{rec}\ \beta.\ b \equiv b[\beta \mapsto \textsc{rec}\ \beta.\ b]$
	R2. $\textsc{rec}\ \beta.\ b \equiv \textsc{rec}\beta'.b[\beta \mapsto \beta']$ if β' not free in $\textsc{rec}\beta.b$

Table 3: Ordering on Behaviours

The *typing judgements* have the form $tenv \vdash e : t \ \&\ b$ where $tenv$ is a type environment mapping identifiers to type schemes, t is the type of e and b is its behaviour. Since CML has a call-by-value semantics there is no behaviour associated with accessing an identifier and therefore the type environment does not contain any behaviour component (except within type schemes). The typing rules are given in Table 2 and are fairly close to the standard ones except that also behaviour information is collected. The types of constants and identifiers are obtained as generic instances of the appropriate type schemes. The actual behaviour is ϵ but, as mentioned earlier, we may want to use a larger behaviour and to express this we introduce an ordering $\sqsubseteq$ on behaviours. This turns out to be a general pattern of the axioms and rules: it is always possible to enlarge the actual behaviour.

In the rule for function abstraction we record the behaviour of the body of the function as the latent behaviour of the function type. The construction of a function does not in itself have an observable behaviour and so is ϵ. In the rule for function application we see that the actual behaviour of the composite construct is that of the operator followed by that of the operand and then the behaviour of the function application itself; the latter is exactly the latent behaviour of the function type. Again we note that this rule is only sound because CML has a call-by-value semantics. In the rule for local definitions we generalise over those type variables, behaviour variables and region variables that neither occur free in the type environment nor in the behaviour; this is expressed by

$$\begin{aligned}&\mathrm{gen}(tenv, b)t = \\ &\quad \text{let } \{\vec{\alpha}\vec{\beta}\vec{\rho}\} = FV(t) \setminus (FV(tenv) \cup FV(b)) \text{ in } \forall\vec{\alpha}\vec{\beta}\vec{\rho}.t\end{aligned}$$

where $FV(\cdots)$ is the set of free type variables, behaviour variables and region variables of the argument. The actual behaviour of the `let`-construct simply expresses that the local value is computed before the body. In the rule for recursive functions we make sure that the actual behaviour is equal to the latent behaviour of the type of the recursive function. The rule for conditional should be straightforward.

The ordering $\sqsubseteq$ on behaviours is defined by the axioms and rules of Table 3. Thus we require $\sqsubseteq$ to be a pre-order and a pre-congruence. We use $\equiv$ for the associated equivalence, i.e. $b_1 \equiv b_2$ stands for $b_1 \sqsubseteq b_2$ and $b_1 \sqsupseteq b_2$, whereas we reserve $=$ for syntactic identity. Furthermore, sequencing is an associative operation with ϵ as identity and we have a distributive law with respect to choice. A consequence of the laws for choice is that it is the least upper bound operator and hence associative and commutative. Finally, the law for recursion allows to unfold the REC-construct and we have a law for alpha-renaming the bound variable.

The main difference between the typing system presented here and those more closely following [12] is that we keep track of the dependencies between the individual communications. If we were to get rid of the difference between sequencing and choice and to allow all behaviours to be implicitly recursive we could move closer to the world of [12, 13] by extending Table 3 with the laws $b_1; b_2 \equiv b_1 + b_2$ and $\textsc{rec}\ \beta.\ b \equiv b[\beta \mapsto \epsilon]$; but then we would loose so much causality that the applications in [10] would no longer be possible.

3 Type and Behaviour Reconstruction

The key idea to our use of algebraic reconstruction is that certain behaviours are replaced by behaviour variables; the information that might get lost by such a replacement is then retained by a collection of constraints that relate the behaviour variables to the behaviours. Since the behaviour variables will be chosen as "fresh" variables in the algorithm this ensures that behaviour variables identify unique behaviour positions — but only until unification forces us to "merge" such positions in the manner alluded to in the discussion of "early subsumption" above.

To carry this idea through we define *simple types*

$$\begin{array}{rcl} s & ::= & \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \mid \alpha \mid s_1 \rightarrow^{\beta} s_2 \mid s_1 \times s_2 \mid s\ \texttt{list} \\ & \mid & s\ \texttt{chan}\ r \mid s\ \texttt{com}\ \beta \end{array}$$

to be types where only behaviour variables occur as latent behaviours of functions and suspended communications. In a similar manner we define *simple behaviours*

$$d \quad ::= \quad \epsilon \mid r!s \mid r?s \mid s\ \textsc{chan}\ r \mid \beta \mid \textsc{fork}\ d \mid d_1; d_2 \mid d_1 + d_2$$

to be behaviours that only use simple types. Additionally we ban the use of (explicit) recursive behaviours since for the purposes of the algorithm they are replaced by sets of constraints (that may be implicitly recursive). Finally to retain the relationship between behaviour variables and behaviours we use *constraints* of the form $d \leq \beta$ and we shall write C for a finite set $\{d_1 \leq \beta_1, \cdots, d_k \leq \beta_k\}$, or $[d_1 \leq \beta_1, \cdots, d_k \leq \beta_k]$, of such constraints. The symbol $\leq$ is a formal symbol closely related to $\sqsubseteq$: a substitution φ solves C, written $\varphi \models C$, iff $\varphi d_i \sqsubseteq \varphi \beta_i$ for all $d_i \leq \beta_i$ in C.

A *constrained type scheme* (or simple scheme) is of the form $\forall \vec{\alpha}\vec{\beta}\vec{\rho}.s[C]$ where the type t of a type scheme is replaced by the "pair" $s[C]$ consisting of a simple type and a set of constraints. Intuitively, the "translation" from t to $s[C]$ proceeds as follows: whenever t has a behaviour $d + \beta$ at a certain position replace it by the behaviour variable β and add the constraint $d \leq \beta$ to C. Several examples of this "translation" is given by comparing the constrained type schemes of Table 4 to the type schemes of Table 1. One subtle point may need to be clarified: we choose to collect $d \leq \beta$ rather than $d + \beta = \beta$

c	CTypeOf(c)
`+`	$\forall\beta_1,\beta_2.\ \texttt{int} \rightarrow^{\beta_1} \texttt{int} \rightarrow^{\beta_2} \texttt{int}\ [\epsilon \le \beta_1, \epsilon \le \beta_2]$
`pair`	$\forall\alpha_1,\alpha_2,\beta_1,\beta_2.\ \alpha_1 \rightarrow^{\beta_1} \alpha_2 \rightarrow^{\beta_2} \alpha_1 \times \alpha_2\ [\epsilon \le \beta_1, \epsilon \le \beta_2]$
`fst`	$\forall\alpha_1,\alpha_2,\beta.\ \alpha_1 \times \alpha_2 \rightarrow^{\beta} \alpha_1\ [\epsilon \le \beta]$
`snd`	$\forall\alpha_1,\alpha_2,\beta.\ \alpha_1 \times \alpha_2 \rightarrow^{\beta} \alpha_2\ [\epsilon \le \beta]$
`send`	$\forall\alpha,\beta_1,\beta_2,\rho.\ (\alpha\ \texttt{chan}\ \rho) \times \alpha \rightarrow^{\beta_1} \alpha\ \texttt{com}\ \beta_2\ [\epsilon \le \beta_1, \rho!\alpha \le \beta_2]$
`receive`	$\forall\alpha,\beta_1,\beta_2,\rho.\ (\alpha\ \texttt{chan}\ \rho) \rightarrow^{\beta_1} \alpha\ \texttt{com}\ \beta_2\ [\epsilon \le \beta_1, \rho?\alpha \le \beta_2]$
`choose`	$\forall\alpha,\beta_1,\beta_2,\beta_3.\ (\alpha\ \texttt{com}\ \beta_1)\ \texttt{list} \rightarrow^{\beta_2} \alpha\ \texttt{com}\ \beta_3\ [\epsilon \le \beta_2, \beta_1 \le \beta_3]$
`wrap`	$\forall\alpha_1,\alpha_2,\beta_1,\beta_2,\beta_3,\beta_4.\ (\alpha_1\,\texttt{com}\,\beta_1) \times (\alpha_1 \rightarrow^{\beta_2} \alpha_2) \rightarrow^{\beta_3} \alpha_2\ \texttt{com}\ \beta_4$ $[\epsilon \le \beta_3, \beta_1;\beta_2 \le \beta_4]$
`sync`	$\forall\alpha,\beta_1,\beta_2.\ (\alpha\ \texttt{com}\ \beta_1) \rightarrow^{\beta_2} \alpha\ [\beta_1 \le \beta_2]$
`channel`	$\forall\alpha,\beta,\rho.\ \texttt{unit} \rightarrow^{\beta} (\alpha\ \texttt{chan}\ \rho)\ [\alpha\ \texttt{CHAN}\ \rho \le \beta]$
`fork`	$\forall\alpha,\beta_1,\beta_2.\ (\texttt{unit} \rightarrow^{\beta_1} \alpha) \rightarrow^{\beta_2} \texttt{unit}\ [\texttt{FORK}\ \beta_1 \le \beta_2]$

Table 4: Constrained Type Schemes for Selected Constants

because this suits us later; however, semantically there is no difference because $d \sqsubseteq \beta$ is equivalent to $d + \beta \equiv \beta$ given that $+$ is the least upper bound operator.

A typical use of the type and behaviour reconstruction algorithm $\mathcal{W}$ upon an expression e takes the form

$$\mathcal{W}\ senv\ e\ =\ (\theta, s, d, C, S)$$

Here *senv* is a *constrained type environment* (or simple environment) that maps identifiers to constrained type schemes. The first result θ is a *simple substitution* that maps type variables to simple types, behaviour variables to behaviour variables (and *not* to simple behaviours), and region variables to region variables. Naturally s is a simple type and d is a simple behaviour. If we had no need to consider constraints we would formulate soundness of $\mathcal{W}$ by stating that $\theta(senv) \vdash e : s\ \&\ d$ is derivable in the inference system of Table 2.

The need to consider constraints presents several complications that are further aggravated by our general treatment of `let` in Table 2 where we use the behaviour information when deciding which variables to generalise (as opposed to trying to use the concept of expansiveness). An obvious need for $\mathcal{W}$ is to collect the set C of constraints. For the statement and proof of soundness we shall be interested in a record of other "phenomena" taking place in the course of execution. One piece of information is the set of "fresh" variables that are generated; we could choose to collect this in the *solution restriction* S but following the usual "sloppiness" dating back to the presentation of the original algorithm $\mathcal{W}$ we shall abstain from this and merely write Fresh(*senv*,*e*) for this set. Another piece of information is a record of all variables generalised in an internal `let`-construct; we shall choose to be precise about this and place the entry G, in S whenever a set , of type, behaviour or region variables has been generalised. A third piece of information is a record of all instantiations of (constrained) type schemes taking place for constants and identifiers; we shall choose to be precise about this and place the entry $\vec{\gamma} : \vec{\gamma}'$ in S whenever $\vec{\gamma}'$ is an instance of $\vec{\gamma}$. Special care must be taken when applying substitutions to solution restrictions. This is done pointwise and we define $\theta(\vec{\gamma} : \vec{\gamma}')$ to be $\vec{\gamma} : \theta\vec{\gamma}'$ and $\theta(G,)$ to be $G,$ for reasons to become clear later.

In the next section we shall formalise soundness of $\mathcal{W}$ based on the idea that $\phi\langle\theta(senv)\rangle \vdash e : \phi(s)\ \&\ \phi(d)$ should be provable whenever ϕ is a solution to C that is faithful to the solution restriction S. The definition of $\mathcal{W}$ is given in Table 5 and is explained in some detail below. Throughout we use the following conventions: the functional composition of the substitutions φ_1 and φ_2 is written $\varphi_1\varphi_2$ rather than $\varphi_1 \circ \varphi_2$; similarly the application of substitution $\varphi_1\varphi_2$ to an argument s is written $\varphi_1\varphi_2 s$ rather than $\varphi_1(\varphi_2 s)$. These conventions bind more tightly than any other operations (e.g. set union or semi-colon).

In the clause for constants we need to produce a new generic instance of the constrained type scheme for the constant. This is accomplished by the function INS defined by

$$\begin{aligned}\text{INS } (\forall\vec{\alpha}\vec{\beta}\vec{\rho}.s[C]) = \text{let } & \vec{\alpha}'\vec{\beta}'\vec{\rho}' \text{ be fresh}\\ & \theta = [\vec{\alpha}\vec{\beta}\vec{\rho} \mapsto \vec{\alpha}'\vec{\beta}'\vec{\rho}']\\ \text{in } & (\theta s)[\theta C], \{\vec{\alpha}\vec{\beta}\vec{\rho} : \vec{\alpha}'\vec{\beta}'\vec{\rho}'\}\end{aligned}$$

It may appear strange that the solution restriction is thrown away; later we shall see that this is correct because no constraint set of Table 4 is implicitly recursive.

For identifiers we have the analogous clause except that the solution restriction is retained; later we shall see that this is necessary because we cannot a priori guarantee that there is no implicit recursion in the constraint set for the identifier.

In the clause for function abstraction the recursive call uses an empty constraint set for the fresh type variable because a type variable contains no behaviours. Since the latent behaviour of the function space should really be d (or perhaps $d + \beta$) it is sensible to use the behaviour variable β provided we add the constraint $d \leq \beta$.

Unification is as normal except that we must take care also to unify behaviour variables and region variables. For type variables we have the usual "occurs check" and we should stress that the set of constraints does not enter into this (unlike [12]); the reason is that we can solve all sets of constraints and so do not need to check for circular behaviours. For reasons of space we omit formal definition of UNIFY.

The interesting part of the clause for `let` is the treatment of generalisation. As an analogue of gen($tenv$,b)t used in Table 2 we here use

$$\begin{aligned}\text{GEN}(senv,\ d)\ s[C] = \text{let } & \{\vec{\alpha}\vec{\beta}\vec{\rho}\} = \mathcal{G}(s, d, senv, C)\\ \text{in } & \forall\vec{\alpha}\vec{\beta}\vec{\rho}.s[C],\ \{G\{\vec{\alpha}\vec{\beta}\vec{\rho}\}\}\end{aligned}$$

for a suitable set $\mathcal{G}(s, d, senv, C)$. Note that we incorporate the entire constraint set in the result of $\mathcal{W}$; the rationale is that even if the `let`-bound variable does not occur in the body we must still be able to type its defining expression. Also note that we do not attempt to reduce the size of the constraint set bound into the constrained type scheme as might be sensible given that the entire set has been incorporated into the result of $\mathcal{W}$; while such reductions may be possible it seems incorrect to simply adapt the constraint-splitting of [12].

To define $\mathcal{G}$ we need an auxiliary concept. A set Y of variables is said to *respect* a set C of constraints if each constraint of C satisfies that it either only involves variables of Y or of the complement of Y:

$$\forall(d \leq \beta) \in C :\ FV(d \leq \beta) \subseteq Y\ \vee\ FV(d \leq \beta) \cap Y = \emptyset$$

In this case C may be split into two disjoint sets C_1 and C_2 such that their union still

```
$\mathcal{W}\ senv\ \ c =$
        let $s[C], S = \mathrm{INS}(\mathrm{CTypeOf}\ c)$
        in $(Id, s, \epsilon, C, \emptyset)$

$\mathcal{W}\ senv\ \ x =$
        let $s[C], S = \mathrm{INS}(senv\ x)$
        in $(Id, s, \epsilon, C, S)$

$\mathcal{W}\ senv\ (\texttt{fn}\ x\ \texttt{=>}\ e) =$
        let $\alpha, \beta$ be fresh
            $(\theta, s, d, C, S) = \mathcal{W}\ (senv[x \mapsto \alpha\ [\ ]])\ e$
        in $(\theta, \theta\alpha \rightarrow^{\beta} s, \epsilon, \{d \leq \beta\} \cup C, S)$

$\mathcal{W}\ senv\ \ (e_1\ e_2) =$
        let $\alpha, \beta$ be fresh
            $(\theta_1, s_1, d_1, C_1, S_1) = \mathcal{W}\ senv\ e_1$
            $(\theta_2, s_2, d_2, C_2, S_2) = \mathcal{W}\ (\theta_1 senv)\ e_2$
            $\theta = \mathrm{UNIFY}\ (\theta_2 s_1, s_2 \rightarrow^{\beta} \alpha)$
        in $(\theta\theta_2\theta_1, \theta\alpha, \theta(\theta_2 d_1; d_2; \beta), \theta\theta_2 C_1 \cup \theta C_2, \theta\theta_2 S_1 \cup \theta S_2)$

$\mathcal{W}\ senv\ (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) =$
        let $(\theta_1, s_1, d_1, C_1, S_1) = \mathcal{W}\ senv\ e_1$
            $ss, S = \mathrm{GEN}(\theta_1 senv, d_1)\ s_1[C_1]$
            $(\theta_2, s_2, d_2, C_2, S_2) = \mathcal{W}\ ((\theta_1 senv)[x \mapsto ss])\ e_2$
        in $(\theta_2\theta_1, s_2, \theta_2 d_1; d_2, \theta_2 C_1 \cup C_2, \theta_2 S_1 \cup \theta_2 S \cup S_2)$

$\mathcal{W}\ senv\ (\texttt{rec}\ f\ x\ \texttt{=>}\ e) =$
        let $\alpha_1, \alpha_2, \beta$ be fresh
            $(\theta, s, d, C, S) = \mathcal{W}\ (senv[f \mapsto \alpha_1 \rightarrow^{\beta} \alpha_2\ [\ ]][x \mapsto \alpha_1\ [\ ]])\ e$
            $\theta' = \mathrm{UNIFY}\ (\theta\alpha_2, s)$
        in $(\theta'\theta, \theta'\theta\alpha_1 \rightarrow^{\theta'\theta\beta} \theta' s, \epsilon, \theta' C \cup \{\theta' d \leq \theta'\theta\beta\}, \theta' S)$

$\mathcal{W}\ senv\ \ (\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2) =$
        let $(\theta, s, d, C, S) = \mathcal{W}\ senv\ e$
            $\theta_0 = \mathrm{UNIFY}\ (s, \texttt{bool})$
            $(\theta_1, s_1, d_1, C_1, S_1) = \mathcal{W}\ (\theta_0\theta senv)\ e_1$
            $(\theta_2, s_2, d_2, C_2, S_2) = \mathcal{W}\ (\theta_1\theta_0\theta senv)\ e_2$
            $\theta' = \mathrm{UNIFY}\ (\theta_2 s_1, s_2)$
        in $(\theta'\theta_2\theta_1\theta_0\theta, \theta' s_2, \theta'\theta_2\theta_1\theta_0 d; (\theta'\theta_2 d_1 + \theta' d_2),$
            $\theta'\theta_2\theta_1\theta_0 C \cup \theta'\theta_2 C_1 \cup \theta' C_2, \theta'\theta_2\theta_1\theta_0 S \cup \theta'\theta_2 S_1 \cup \theta' S_2)$
```

Table 5: Type and Behaviour Reconstruction

gives C but C_1 only involves variables of Y and C_2 does not involve variables of Y. To prepare for the treatment of soundness, in particular the existence of solutions faithful to the solution restriction, we demand that $\mathcal{G}(s, d, senv, C)$ respects C. As was the case for gen we want $\mathcal{G}(s, d, senv, C)$ to be disjoint with $FV(d) \cup FV(senv)$. Surely $\mathcal{G}(s, d, senv, C)$ should contain variables of s but since much of the information that "ought to be" in s has been placed in C we should take this into account by "closing under C". Formally we define the closure of a set X under C by the formula

$$X^C \ = \ \{\gamma_n \mid \gamma_0 \in X \wedge \forall i < n : (\cdots\gamma_{i+1}\cdots \leq \gamma_i) \in C\}$$

(reminiscent of the definition of the set of sentential forms generated by the nonterminals X given the grammar C). We then define $\mathcal{G}(s, d, senv, C)$ to be the maximal set fulfilling these restrictions:

$$\mathcal{G}(s, d, senv, C) = \bigcup\{Y \mid Y \subseteq FV(s)^C \backslash (FV(d) \cup FV(senv)),\ Y \text{ respects } C\}$$

Lemma 3.1 This defines a set that respects C.

Variations in the definition of $\mathcal{G}$ are of course conceivable. An obvious question is whether also to close $FV(d) \cup FV(senv)$ under C following [6]; but given our insistence on "respecting C" this does not affect the definition of $\mathcal{G}$.

The clauses for recursion and conditional should present no surprises.

Fact 3.2 $\mathcal{W}$ always terminates (possibly with failure).

Existence of solutions

Having constructed a (necessarily finite) set of constraints our next task is to solve them. Due to the presence of recursive behaviours we can modify the approach of [12] to construct solutions to all finite sets of constraints. Formally a solution ϕ to a set of constraints C is a substitution that maps behaviour variables to behaviours (*not* simple behaviours only) such that $\phi d \sqsubseteq \phi\beta$ holds for all $d \leq \beta$ in C. The *canonical solution* $\overline{C}$ is constructed as follows:

- if $C = \emptyset$ then

  $$\overline{C} = Id$$

  where Id is the identity substitution
- if $d_1 \leq \beta$, $\cdots$, $d_k \leq \beta$ are all the constraints in C with right-hand side β then

  $$\overline{C} = [\beta \mapsto \textsc{rec}\ \beta.\overline{C_\beta}(d_1 + \cdots + d_k)] \circ \overline{C_\beta}$$

  where C_β is $C \setminus \{d_1 \leq \beta, \cdots, d_k \leq \beta\}$.

There is a fair amount of nondeterminacy in this definition. We may reduce this by assuming a linear order on behaviour variables and then consider the behaviour variables in increasing order. The final ambiguity is the order in which the $d_1, \cdots, d_k$ are collected but this does not influence the solution (modulo $\equiv$).

Lemma 3.3 All (finite) sets of constraints have a solution; in particular: $\overline{C} \models C$.

The difficulty with this canonical solution is to find a sense in which it is "principal". To illustrate this consider the constraint $\beta \leq \beta$. Its canonical solution is $[\beta \mapsto \textsc{rec}\beta.\beta]$ but $[\beta \mapsto \beta]$, $[\beta \mapsto \epsilon]$, $[\beta \mapsto \rho!\alpha]$ and $[\beta \mapsto \textsc{rec}\beta.(\beta + \rho!\alpha)]$ are also solutions. To define "principality" we need to define an operator $\triangleright$ such that $\beta \triangleright \textsc{rec}\beta.\beta$, $\epsilon \triangleright \textsc{rec}\beta.\beta$, $\rho!\alpha \triangleright \textsc{rec}\beta.\beta$ and $\textsc{rec}\beta.(\beta + \rho!\alpha) \triangleright \textsc{rec}\beta.\beta$. It is by no means clear how to use $\sqsubseteq$ and composition of substitutions to achieve this. (It is easy to prove that $\sqsupseteq$ cannot be used for $\rhd$.) This then motivates our design of an algorithm that does not *assume* the existence of "principal" solutions.

Note that the use of $\textsc{rec}\beta$ in the definition of $\overline{C}$ above is superfluous if the $\overline{C_\beta}(d_1+\cdots+d_k)$ in question does not contain β: simply use **R1** to unfold $\textsc{rec}\beta.\overline{C_\beta}(d_1+\cdots+d_k)$ to $\overline{C_\beta}(d_1+\cdots+d_k)$. This is the case if C contains no implicit recursion over behaviour variables. Then much as in [12] any solution ϕ to $\{d_1 \leq \beta, \cdots, d_k \leq \beta\}$ satisfies $\phi\beta \sqsupseteq \phi(d_1+\cdots+d_k) \sqsupseteq \phi(\overline{C_\beta}(d_1+\cdots+d_k)) \equiv \phi(\overline{C}\beta)$ so in a sense $\overline{C}\beta$ is "principal" (least) in this case: in [12] $\phi \models C$ implies $\phi = \phi\circ\overline{C}$ whereas we get $\phi \sqsupseteq \phi\circ\overline{C}$ (provided C has no implicit recursion!). These remarks shed some additional light on our different treatment of constants and identifiers in algorithm $\mathcal{W}$.

4 Soundness

To state the soundness of algorithm $\mathcal{W}$ we need a few auxiliary concepts and notations and to conduct the proof we need a technical proposition about the algorithm. We therefore begin by establishing the required terminology.

Recall our notation for substitutions: we use φ (and φ', φ_1 etc.) for general substitutions, θ (and θ', θ_1 etc.) for simple substitutions, and ϕ (and ϕ', ϕ_1 etc.) for solution substitutions. The domain $\mathrm{Dom}(\varphi)$ of a substitutions φ is the finite set of variables not mapped to themselves. We shall say that φ *has no effect* on a type s if $\mathrm{Dom}(\varphi) \cap FV(s) = \emptyset$ so that $\varphi s = s$; similar terminology applies to behaviours and constraints. The substitution φ *involves* the set $Inv(\varphi) = \mathrm{Dom}(\varphi) \cup \bigcup\{\, FV(\varphi\gamma) \mid \gamma \in \mathrm{Dom}(\varphi) \,\}$ of variables; we shall say that φ does *not involve* a type s if $Inv(\varphi) \cap FV(s) = \emptyset$ and similarly for behaviours, constraints, sets of variables, etc. For types, behaviours and constraints we shall say that the set of variables involved are their set of free variables. The application of φ to a simple scheme $\forall\vec{\gamma}.s[C]$ is $\forall\vec{\gamma}'.(\varphi\circ[\vec{\gamma} \mapsto \vec{\gamma}'])(s[C])$ where $\vec{\gamma}'$ is chosen so as to avoid capture of free variables; there is some amount of nondeterminacy in this definition and we shall assume that $\vec{\gamma}'$ is chosen to be $\vec{\gamma}$ whenever possible. The application of a substitution may be extended to a simple environment in a "pointwise" manner.

The set $BV(\forall\vec{\gamma}.s[C])$ of bound variables of $\forall\vec{\gamma}.s[C]$ is $\{\vec{\gamma}\}$ and the set $FV(\forall\vec{\gamma}.s[C])$ of free variables is $FV(s[C])\backslash\{\vec{\gamma}\}$. For a simple environment *senv* we define $BV(senv) = \bigcup\{BV(senv(x)) \mid x \text{ in the domain of } senv\}$ and similarly for $FV(senv)$. It is possible that a variable occurs free in *senv* as well as bound at more than one identifier; we shall say that *senv* is *consistent* if this is not the case:

$$\forall x{\neq}y\colon\ BV(senv(x)) \cap (BV(senv(y)) \cup FV(senv(y))) = \emptyset$$

(Consistency can always be achieved by alpha-renaming.) Next, we write $GV(S)$ for the set $\bigcup\{,\ \mid G,\ \in S\}$ of variables recorded to be generalised in the solution restriction S, $LV(S)$ for the set $\{\gamma \mid (\cdots\gamma\cdots : \cdots\gamma'\cdots) \in S\}$ of instantiated variables and $RV(S)$ for the set $\{\gamma' \mid (\cdots\gamma\cdots : \cdots\gamma'\cdots) \in S\}$ of instantiation variables.

We also need to be able to keep the different "classes" of polymorphic variables separate: to this end we define $\mathcal{BV}(senv) = \{BV(senv(x)) \mid x \text{ in the domain of } senv\}$ and $\mathcal{GV}(S) = \{,\ \mid G,\ \in S\}$. Note that $BV(senv) = \bigcup\mathcal{BV}(senv)$ and $GV(senv) = \bigcup\mathcal{GV}(senv)$ and that consistency of *senv* implies that the sets of $\mathcal{BV}(senv)$ are mutually disjoint.

We can now strengthen Fact 3.2 by stating a technical proposition about the result (θ, s, d, C, S) produced by $\mathcal{W}$ *senv e* for consistent *senv*: e.g. that θ is a simple and idempotent substitution. For reasons of space we omit the formal statement.

We now turn towards the statement of soundness. Recall that a substitution ϕ solves a set C of constraints, written $\phi \models C$, if $\phi d \sqsubseteq \phi\beta$ holds for all $d \leq \beta$ in C. It satisfies a simple scheme ($\phi \models \forall\vec{\gamma}.s[C]$) if it satisfies the constraints embedded ($\phi \models C$). Note that for the purpose of satisfying simple schemes we do *not* regard the quantified variables as concealed from the outside; this creates no problems because we work with consistent environments that do not involve freshly generated variables. We also need a special notion of applying a substitution ϕ to a simple scheme $\forall\vec{\gamma}.s[C]$ in order to obtain a type scheme:

$$\phi\langle\forall\vec{\gamma}.s[C]\rangle = \forall\gamma' \in FV(\phi\vec{\gamma}).\ \phi(s)$$

This definition would be problematic if $FV(\phi\vec{\gamma}) \setminus \{\vec{\gamma}\}$ contains a variable free in s because then this variable gets bound as part of the application. Rather than amending this by replacing $FV(\phi\vec{\gamma})$ with e.g. $FV(\phi\vec{\gamma})\setminus(FV(s) \setminus \{\vec{\gamma}\})$ we shall make sure only to work with solution substitutions where this phenomena does not occur. The special notion of application may be extended to simple environments in a "pointwise" manner.

One restriction we impose upon a solution ϕ is that it maintains the distinction between "polymorphic variables", i.e. $BV(senv) \cup GV(S)$, and "free variables", i.e. $FV(senv)\cup\text{Fresh}(senv, e)$. Additionally a solution must maintain the distinction between the different "groups" of polymorphic variables. To express this concisely write

$$\beta_1 \#_\phi \beta_2 \text{ iff } FV(\phi\beta_1) \cap FV(\phi\beta_2) = \emptyset$$

and define

$$\begin{aligned}\phi \ \ddagger\ \mathcal{P}, F \text{ iff the sets of } \mathcal{P} \text{ are mutually disjoint and}\\ \forall\beta_1,\beta_2 \in (\textstyle\bigcup\mathcal{P}) \cup F : \beta_1 \#_\phi \beta_2 \ \vee\ \{\beta_1,\beta_2\} \subseteq F \setminus (\textstyle\bigcup\mathcal{P}) \ \vee\\ \exists P \in \mathcal{P} : \{\beta_1,\beta_2\} \subseteq P\end{aligned}$$

The condition then is that $\phi \ \ddagger\ \mathcal{BV}(senv) \cup \mathcal{GV}(S), FV(senv)\cup\text{Fresh}(senv, e)$ and then the capture of free variables in $\phi\langle\cdots\rangle$ cannot take place. Note that $\phi \ddagger \mathcal{P}, F$ is equivalent to $\phi \ \ddagger\ \mathcal{P}, F \setminus \bigcup\mathcal{P}$ as well as $\phi \ \ddagger\ \mathcal{P} \cup \{F \setminus \bigcup\mathcal{P}\}, \emptyset$.

Another restriction we need to impose upon a solution ϕ is due to the fact that we do not assume the existence of principal solutions. Yet it will be of importance for soundness that the solutions to a generic scheme is "compatible" with the solutions to its instantiations. To this end define

$$\begin{aligned}\phi \dagger S \text{ iff } \forall(\vec{\alpha}\vec{\beta}\vec{\rho} : \vec{\alpha}'\vec{\beta}'\vec{\rho}') \in S : \text{each } \phi\beta' \equiv b \text{ for some } b \text{ that is a substitution}\\ \text{instance of } \phi\beta\end{aligned}$$

so that $\phi \dagger S$ expresses the desired "compatibility" (modulo $\equiv$).

To prepare for the statement of soundness we write $tenv \vdash e : t \ \&\ b$ (modulo $\equiv$) as a shorthand for: there exists $tenv'$, t' and b' such that $tenv' \vdash e : t' \ \&\ b'$ and $tenv \equiv tenv'$, $t \equiv t'$ and $b \equiv b'$. We then have

Theorem 4.1 Suppose $\mathcal{W}$ *senv e* $= (\theta, s, d, C, S)$ and *senv* is consistent. Then

- $\phi \models C \wedge \forall x : \phi \models \theta(senv(x))$
- $\phi \dagger S \wedge \phi \ \ddagger\ \mathcal{BV}(senv) \cup \mathcal{GV}(S),\ FV(senv)\cup\text{Fresh}(senv, e)$

implies $\phi\langle\theta\ senv\rangle \vdash e : \phi(s)\ \&\ \phi(d)$ (modulo $\equiv$).

The condition that $\phi \models C$ simply states that ϕ solves the constraints generated. But in order that the translation of the simple environment *senv* to the type environment $\phi\langle\theta senv\rangle$ is meaningful we need to ensure that also the constraints in the simple type environment are correctly solved. We could have used a "solution environment" for this but our assumptions on consistency etc. ensure that we can dispense with this machinery. The two conditions on faithfulness with respect to S have already been motivated.

The proof of this result uses a number of observations. For these we define $ts \succeq ts'$ (modulo $\equiv$) to mean that if $ts \succ t'$ then there exists t such that $ts \succ t$ and $t \equiv t'$.

Fact 4.2 $b_1 \sqsupseteq b_2$ is equivalent to $b_1 \equiv b_1 + b_2$ and implies $FV(b_1) \supseteq FV(b_2)$.

Lemma 4.3 If $\phi \models C$ then $\bigcup\{\ FV(\phi\gamma) \mid \gamma \in X^C\} = \bigcup\{\ FV(\phi\gamma) \mid \gamma \in X\}$ for all sets X.

Lemma 4.4 For all constants c of Tables 1 and 4: if $\phi \models \mathrm{CTypeOf}(c)$ then $\mathrm{TypeOf}(c) \succeq \phi\langle\mathrm{CTypeOf}(c)\rangle$ (modulo $\equiv$).

Existence of solutions

The conditions imposed upon the solution ϕ in the statement of soundness imply that even when the simple environment *senv* is empty it will not always work to use the canonical solution $\overline{C}$ to the set C of constraints produced. The reason is that $\overline{C}$ need not be faithful to the solution restriction S, in particular $\overline{C} \dagger S$ might fail.

To sketch this consider distinct behaviour variables β_1 and β_2 and constraints $b_1 \leq \beta_1$ and $b_2 \leq \beta_2$ where we shall assume that b_1 and b_2 have no free variables and that we do not have $b_1 \sqsubseteq b_2$ nor $b_1 \sqsupseteq b_2$. Let β_1 and β_2 be polymorphic variables and suppose they are instantiated to fresh behaviour variables β_1' and β_2'; we then generate the constraints $b_1 \leq \beta_1'$ and $b_2 \leq \beta_2'$. If we later unify β_1' and β_2' we intuitively get the constraints $b_1 + b_2 \leq \beta_1'$ and $b_1 + b_2 \leq \beta_2'$. Thus the canonical solution (modulo $\equiv$, in particular after using **R1**) maps β_1 to b_1 but β_1' to $b_1 + b_2$ and similarly for β_2 and β_2'. This contradicts the presence of $(\beta_1 : \beta_1')$ in S since $b_1 + b_2$ is not a substitution instance of b_1 (modulo $\equiv$).

Luckily we can overcome the above problem by accepting canonical solutions to modified sets of constraints than those produced by the algorithm. To prepare for this we shall say that *senv respects itself* if all $senv(x) = \forall\vec{\gamma_x}.s_x[C_x]$ satisfy that $\{\vec{\gamma_x}\}$ respects C_x.

Theorem 4.5 Let *senv* be consistent and respect itself and suppose that $\mathcal{W}\ senv\ e = (\theta, s, d, C, S)$. Then there exists a solution ϕ such that

- $\phi \models C \wedge \forall x : \phi \models \theta(senv(x))$
- $\phi \dagger S \wedge \phi \ddagger \mathcal{BV}(senv) \cup \mathcal{GV}(S),\ FV(senv) \cup \mathrm{Fresh}(senv, e)$.

5 Conclusion

Our treatment of `let` is necessarily more complex than in [4] because we want to obtain more polymorphism than is possible using the concept of expansiveness: this is a syntactic condition on an expression that guarantees the effect of that expression to be ϵ. In [4] the construct `let` $x = e_1$ `in` e_2 is treated as $e_2[e_1/x]$ for expansive e_1 (thus admitting polymorphism) and as (`fn` x `=>` e_2)(e_1) for non-expansive e_1 (thus preventing polymorphism). We wish to achieve polymorphism even for non-expansive e_1. So suppose $e_1 = e_{11}\,;e_{12}$ where only e_{12} has effect equivalent to ϵ. Then we shall treat `let` $x = e_{11}\,;e_{12}$ `in` e_2 as $e_{11}\,;e_{12}\,;(e_2[e_{12}/x])$. In general we cannot syntactically dissect e_1 into its "expansive" part e_{12} and its "non-expansive" part e_{11} and we therefore follow [12] in the more complicated definition of generalisation.

Our behaviours have much more algebraic structure than the effects of [12] and the presence of recursive behaviours means that we can solve all constraints. The price to pay for the latter is that the canonical solution does not appear to be principal and consequently our algorithm deviates fundamentally from [12]. This may be paraphrased by saying that [12] solves constraints on-the-fly (to facilitate the "occurs check" and the splitting of constraints) whereas our algorithm involves no solving on-the-fly.

To investigate the link between unification in non-free algebras and algebraic reconstruction (in the sense of [12]) consider the types $t_1 = \texttt{int} \rightarrow^{\rho_1!\alpha_1+\beta_1} \texttt{int}$ and $t_2 = \texttt{int} \rightarrow^{\rho_2?\alpha_2+\beta_2} \texttt{int}$. It is impossible to unify them if behaviours constitute a free algebra because $\rho_1!\alpha_1$ cannot be unified with $\rho_2?\alpha_2$. If behaviours admit commutativity for $+$, as is the case given the axiomatisation of Table 3, we can rearrange terms and use the substitution $\theta = [\beta_1 \mapsto \rho_2?\alpha_2,\ \beta_2 \mapsto \rho_1!\alpha_1]$; more precisely we perform unification modulo $\equiv$, i.e. we get $\theta t_1 \equiv \theta t_2$, where $\equiv$ has been extended from behaviours to types in the obvious manner. Another substitution that will unify t_1 and t_2 (modulo $\equiv$) is $\theta' = [\beta_1 \mapsto \rho_2?\alpha_2 + \beta_{12},\ \beta_2 \mapsto \rho_1!\alpha_1 + \beta_{12}]$; this is a "strange" substitution to suggest as the result of unification involves a behaviour variable that is not free in the arguments. We now show that this is essentially what goes on in algebraic reconstruction.

To see this note that using constraints we represent t_1 as $s_1[C_1]$ where $s_1 = \texttt{int} \rightarrow^{\beta_1} \texttt{int}$ and $C_1 = \{\rho_1!\alpha_1 \leq \beta_1\}$ and similarly for t_2. Algebraic reconstruction then "unifies" $s_1[C_1]$ and $s_2[C_2]$ by constructing a unifier θ'' for s_1 and s_2 and by collecting the constraints $C_{12} = \{\rho_1!\alpha_1 \leq \theta''\beta_1,\ \rho_2?\alpha_2 \leq \theta''\beta_2\}$. The substitution θ'' is going to be one of $[\beta_1 \mapsto \beta_2]$ or $[\beta_2 \mapsto \beta_1]$ but let us pretend that instead it is $[\beta_1 \mapsto \beta_{12},\ \beta_2 \mapsto \beta_{12}]$. Then C_{12} boils down to $\{\rho_1!\alpha_1 \leq \beta_{12},\ \rho_2?\alpha_2 \leq \beta_{12}\}$ which under the laws of Table 3 is equivalent to $\{\rho_1!\alpha_1 + \rho_2?\alpha_2 \leq \beta_{12}\}$ and $\{\rho_1!\alpha_1 + \rho_2?\alpha_2 + \beta_{12} = \beta_{12}\}$. This shows that $\theta'' s_i[C_i]$ is just another way of presenting $\theta' t_i$ above.

It thus appears that algebraic reconstruction is a very special case of unification where only the laws for $+$ being least upper bound are of importance and where all substitutions produced have a special form: there always is a "$+\beta_{12}$" component and it is the β_{12} that will be used for further unifications.

An interesting question left open by our treatment is the completeness of $\mathcal{W}$ wrt. the inference system of Table 2. While this question is of great theoretical importance it is in good accord with the considerations of program analysis that we do not regard it as prominently as soundness: the whole nature of program analysis is to trade a lack of precision for acceptable time- and space-complexities — not least because most program analysis questions turn out to be undecidable when exact answers are required.

On the practical side we have experimented with an implementation of (a precurser of) the algorithm presented. This has proven to be a most effective way of increasing one's understanding of the communication behaviour of CML-programs.

Acknowledgements to Torben Amtoft, Fritz Henglein, Pierre Jouvelot, Jean-Pierre Talpin, and Mads Tofte for interesting discussions and to LOMAPS (Esprit Basic Research) and DART (Danish Science Research Council) for funding.

References

[1] B. Berthomieu, T. Le Sergent: Programming with Behaviours in an ML Framework, the Syntax and Semantics of LCS. Proc. ESOP'94, SLNCS 788, 1994.

[2] A. Giacalone, P. Mishra, S. Prasad: Operational and Algebraic Semantics for Facile: A Symmetric Integration of Concurrent and Functional Programming. Proc. ICALP'90, SLNCS 443, 1990.

[3] T. Jensen: Disjunctive Strictness Analysis. Proc. LICS'92, 1992.

[4] P. Jouvelot, D. K. Gifford: Algebraic Reconstruction of Types and Effects. Proc. POPL'90, 1990.

[5] T.-M. Kuo, P. Mishra: Strictness Analysis: A New Perspective based on Type Inference. Proc. FPCA'89, ACM Press, 1989.

[6] X. Leroy, P. Weiss: Polymorphic Type Inference and Assignment. Proc. POPL'90, ACM Press, 1990.

[7] D. Matthews: A Distributed Concurrent Implementation of Standard ML. Proc. EurOpen Autumn 1991 Conference, 1991.

[8] J. C. Mitchell: Type Inference with Simple Subtypes. Journal of Functional Programming **1**, 1991.

[9] F. Nielson, H.R. Nielson: From CML to Process Algebras. Proc. CONCUR'93, SLNCS 715, 1993.

[10] H. R. Nielson, F. Nielson: Higher-Order Concurrent Programs with Finite Communication Topology. Proc. POPL'94, ACM Press, 1994.

[11] J.H. Reppy: Higher-Order Concurrency. Ph.D.-Thesis, Rep. 92-1285, Department of Computer Science, Cornell University, 1992.

[12] J.-P. Talpin, P. Jouvelot: The Type and Effect Discipline. Proc. LICS'92, 1992. (Also see *Information and Computation* **111** *2*, 1994.)

[13] B. Thomsen: Polymorphic sorts and types for concurrent functional programs. Techn. Rep. ECRC-93-10, 1993.

[14] J. H. Siekmann: Unification Theory. Journal of Symbolic Computation **7**, 1989.