Type and Behaviour Reconstruction for Higher-Order Concurrent Programs

Torben Amtoft, Flemming Nielson, Hanne Riis Nielson
DAIMI, Aarhus University
Ny Munkegade, DK-8000 Århus C, Denmark
{tamtoft,fn,hrn}@daimi.aau.dk

Abstract: In this paper we develop a sound and complete type and behaviour inference algorithm for a fragment of CML (Standard ML with primitives for concurrency). Behaviours resemble terms of a process algebra and yield a concise presentation of the communications taking place during execution; types are mostly as usual except that function types and "delayed communication types" are labelled by behaviours expressing the communications that will eventually take place. The development of the present paper improves a previously published algorithm in not only achieving a sound algorithm for type inference but also one that is complete, due to an alternative strategy for generalising over types and behaviours. Although we show how to solve special kinds of constraints the problem of a general solution procedure for the constraints generated remains; this is related to an open problem concerning whether all programs typable in Standard ML (with concurrency primitives) are also typable in our system.

1 Introduction

It is well-known that testing can only demonstrate the presence of bugs, never their absence. This has motivated a vast amount of software related activities on guaranteeing statically (that is, at compile-time rather than run-time) that the software behaves in certain ways; a prime example is the formal (and usually manual) verification of software. In this line of activities various notions of type systems have been put forward because these allow static checks of certain kinds of bugs: while at run-time there may still be a need to check for division by zero there will never be a need to check for the addition of booleans and files. As programming languages evolve in terms of features, like module systems and the integration of different programming paradigms, the research on "type systems" is constantly pressed for new problems to be treated.

Our research has been motivated by the integration of the functional and concurrent programming paradigms. Example programming languages are CML [Rep91] that extends Standard ML with primitives for concurrency, Facile [PGM90] that follows a similar approach but more directly contains syntax for expressing CCS-like process composition, and LCS [BS94]. The overall communication structure of such programs may not be immediately clear and hence one would like to find compact ways of recording the communications taking place during execution. One such representation is *behaviours*, a kind of process algebra expressions.

In [NN93] and [NN94a] inference systems are developed that extend the usual notion of types with a formal way of linking behaviours to the syntax of programs. Applications of such information are demonstrated in [NN94a] and [NN94c].

The question remains: how to implement the inference system, i.e. how to reconstruct the types and behaviours? It seems suitable to use a modified version of algorithm W [Mil78]. This algorithm works by unification, but since our behaviours constitute a non-free algebra (due to the laws imposed on them) this approach is not immediately feasible in our framework. Instead we employ the technique of algebraic reconstruction [JG91, TJ92a]; that is the algorithm unifies the free part of the type structure and generates constraints to cater for the non-free parts.

This idea is carried out in [NN94b], where a reconstruction algorithm is presented which is sound but not complete. The algorithm returns two kind of constraints: C-constraints and S-constraints. The C-constraints represent the "monomorphic" aspects of the analysis whereas the S-constraints are needed to cope with polymorphism: they express that instances of polymorphic variables should remain instances even after applying a solution substitution. Using S-constraints is *not* a standard tool when analysing polymorphic languages; they seem to be needed because the C-constraints apparently lack a "principal solution property" (a phenomenon well-known in unification theory). Finding a solution to C-constraints is feasible as shown in [NN94b]. It is harder to solve S-constraints and in fact they can be seen as a special case of semi-unification; the relationship between type inference for polymorphic languages and semiunification is discussed in [Hen93].

The purpose of the present paper is to present a sound and *complete* reconstruction algorithm. This algorithm also returns S-constraints together with the C-constraints, albeit in a different version than [NN94b]. However, the problem of finding an algorithm for solving these constraints remains.

Overview of paper

In Section 2 we introduce the basic concepts; apart from the treatment of type schemes and from the introduction of S-constraints and from the notion of "downwards closure" they are pretty standard. In Section 3 we present our version of the inference system from [NN94a]. In Section 4 the reconstruction algorithm W is presented. In Section 5 the algorithm is shown to be sound. In Section 6 we motivate our non-standard notion of type schemes, leading to a slightly different (but equivalent) inference system. This inference system allows us to prove the completeness of W in Section 7. In Section 8 we discuss our choice of generalisation strategy for W and compare it to the one used in [NN94b]. Then in Section 9 we show how to solve the constraints generated for some special cases; this solution in effect loses all polymorphism. Section 10 concludes.

2 Preliminaries

CML expressions. Expressions e are built from identifiers x, constants c, applications $e_1\ e_2$, monomorphic abstractions $\lambda x.e_0$, polymorphic abstractions $\mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0$, conditionals $\mathsf{if}\ e_0\ e_1\ e_2$ and recursive constructs $\mathsf{rec}\ f(x) \Rightarrow e_0$.

This is much like ML, the concurrent aspects being taken care of by the constants c. Some of these will be listed in Section 3 together with their types. We refer to [NN94a] for a more thorough motivation.

Free and bound variables. We shall employ two kinds of variables, and use γ to range over either kind: behaviour variables β and type variables α. We write $\mathsf{fv}(\_)$ for the free variables and $\mathsf{bv}(\_)$ for the bound variables – the definition of these sets is mostly obvious. It is convenient also to define $\mathsf{var}(\_) = \mathsf{fv}(\_) \cup \mathsf{bv}(\_)$ and to use F to range over sets of free variables.

Substitutions. These are total functions that for each β produces a behaviour b and for each α produces a type t. We use g to range over types and behaviours collectively; similarly the letters ϕ, ψ and θ range over substitutions. The result of applying a substitution ϕ to some entity g is denoted $g[\phi]$. The substitution which works by first applying ϕ_1 and then applying ϕ_2 is denoted $\phi_1;\phi_2$. The substitution which maps γ into g and leaves all other variables unchanged is denoted $[\gamma \mapsto g]$. If $\gamma[\phi_1] = \gamma[\phi_2]$ for all $\gamma \in F$ we write $\phi_1 \stackrel{F}{=} \phi_2$. The following standard conventions apply for a substitution ϕ: the domain $\mathsf{dom}(\phi) = \{\gamma \mid \gamma[\phi] \neq \gamma\}$; the range $\mathsf{ran}(\phi) = \{\gamma \mid \exists \gamma' \in \mathsf{dom}(\phi) : \gamma \in \mathsf{fv}(\gamma'[\phi])\}$; and $\mathsf{fv}(\phi) = \mathsf{dom}(\phi) \cup \mathsf{ran}(\phi)$.

Behaviours. Behaviours b are built using the syntax

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

that is they can be one of the following: a behaviour variable β; the empty behaviour ϵ (no communications take place); an output action $!t$ (a value of type t is sent); an input action $?t$ (a value of type t is received); a channel action $t\ \mathsf{CHAN}$ (a channel able to transmit values of type t is created); a fork action $\mathsf{FORK}\ b$ (a process with behaviour b is created); a recursive behaviour $\mathsf{REC}\beta.b$ (where b can "call" itself recursively via β); a sequential composition $b_1;b_2$ (first b_1 is performed and then b_2); a non-deterministic choice $b_1 + b_2$ (either b_1 or b_2 are performed).

For a behaviour b we define $\mathsf{bv}(b) = \emptyset$ whereas $\mathsf{fv}(b)$ is defined such that $\mathsf{fv}(\mathsf{REC}\beta.b') = \mathsf{fv}(b') \setminus \{\beta\}$. In other words, the β in $\mathsf{REC}\beta.b'$ is considered a "nameless dummy", i.e. we have some sort of implicit deBruijn-notation. Accordingly we do not distinguish between behaviours which are equivalent modulo alpha-conversion, so $\mathsf{REC}\beta.b$ is considered identical to $\mathsf{REC}\beta'.b[\beta \mapsto \beta']$ if β' does not occur in b (in [NN94b, Table 3] this was listed as axiom R2). Thus it makes sense to define $(\mathsf{REC}\beta.b)[\psi] = \mathsf{REC}\beta.b[\psi]$ where $\beta \notin \mathsf{var}(\psi)$.

Just as in [NN94a, Table 3] we impose a preorder $\sqsupseteq$ on behaviours. The intuition behind $b_1 \sqsupseteq b_2$ is that any action performed by b_2 can be performed by b_1 as well (but not necessarily vice-versa). This preorder is listed in Fig. 1, where $b_1 \equiv b_2$ denotes that $b_1 \sqsupseteq b_2$ and $b_2 \sqsupseteq b_1$.

P1	$b \sqsupseteq b$	**S1**	$b_1;(b_2;b_3) \equiv (b_1;b_2);b_3$
P2	$b_1 \sqsupseteq b_2 \ \wedge\ b_2 \sqsupseteq b_3 \ \Rightarrow\ b_1 \sqsupseteq b_3$	**S2**	$(b_1+b_2);b_3 \equiv (b_1;b_3)+(b_2;b_3)$
C1	$b_1 \sqsupseteq b_2 \ \wedge\ b_3 \sqsupseteq b_4 \ \Rightarrow\ b_1;b_3 \sqsupseteq b_2;b_4$	**E1**	$b \equiv \epsilon; b$
C2	$b_1 \sqsupseteq b_2 \ \wedge\ b_3 \sqsupseteq b_4 \ \Rightarrow\ b_1+b_3 \sqsupseteq b_2+b_4$	**E2**	$b;\epsilon \equiv b$
C3	$b_1 \sqsupseteq b_2 \ \Rightarrow\ \mathsf{FORK}\ b_1 \sqsupseteq \mathsf{FORK}\ b_2$	**J1**	$b_1+b_2 \sqsupseteq b_1 \ \wedge\ b_1+b_2 \sqsupseteq b_2$
C4	$b_1 \sqsupseteq b_2 \ \Rightarrow\ \mathsf{REC}\beta.b_1 \sqsupseteq \mathsf{REC}\beta.b_2$	**J2**	$b \sqsupseteq b+b$
R1	$\mathsf{REC}\beta.b \equiv b[\beta \mapsto \mathsf{REC}\beta.b]$		

Figure 1: The preorder $\sqsupseteq$ on behaviours with equivalence $\equiv$.

Fact 2.1 If $b_1 \sqsupseteq b_2$ then $\mathsf{fv}(b_1) \supseteq \mathsf{fv}(b_2)$, and for any ψ we also have $b_1[\psi] \sqsupseteq b_2[\psi]$.

Types. Types t can be either a type variable α, a base type like Int or Bool or Unit, a function type $t_1 \rightarrow^b t_2$ (given a value of type t_1 a computation is initiated that behaves as indicated by b and that returns a value of type t_2), a list type t **list**, a communication type t **com** b (if such a communication possibility if activated it behaves as indicated by b and returns a value of type t), or a channel type t **chan** (a channel able to transmit values of type t).

For all types t we have $\mathsf{bv}(t) = \emptyset$; so as is the case for behaviours there is no difference between writing $\mathsf{fv}(t)$ ($\mathsf{fv}(b)$) and writing $\mathsf{var}(t)$ ($\mathsf{var}(b)$).

Type schemes. These are denoted by ts and are of form $\forall \overline{F}.t$ where F are the *free* variables: to be more precise we have $\mathsf{fv}(ts) = F$ and $\mathsf{bv}(ts) = \mathsf{fv}(t) \setminus F$. If $F \subseteq \mathsf{fv}(t)$ we say that ts is a *kernel* type scheme; if this is not necessarily the case we say that ts is an *enriched* type scheme. For every ts we define a kernel type scheme $\kappa(ts)$ by $\forall \overline{F'}.t$ where $F' = F \cap \mathsf{fv}(t)$.

> *Remark:* These definitions are highly non-standard. Although there is a natural bijection between "standard" type schemes (of form $\forall \vec{\gamma}.t$ where $\vec{\gamma}$ is the list of bound variables) and kernel type schemes we shall have to deal with enriched type schemes, as will be explained in Section 6.
>
> Further notice that the names of the bound variables *do* matter; they are not just nameless dummies (as they are recorded in S-constraints): so $\forall \emptyset.\alpha \rightarrow^\epsilon \alpha$ is not interchangeable with $\forall \emptyset.\alpha' \rightarrow^\epsilon \alpha'$.

Substitutions on type schemes. Given a type scheme $ts = \forall \overline{F}.t$ and a substitution ψ, we define $ts[\psi] = \forall \overline{F'}.(t[\psi])$ where $F' = \mathsf{var}(F[\psi])$. With this definition we have

$$\mathsf{fv}(ts[\psi]) = \mathsf{var}(\mathsf{fv}(ts)[\psi])$$

but it does *not* necessarily hold that $\mathsf{bv}(ts[\psi]) = \mathsf{var}(\mathsf{bv}(ts)[\psi])$; the latter situation arises when $\beta_1[\psi] = \beta_2[\psi]$ where $\beta_1 \in \mathsf{fv}(ts)$ and $\beta_2 \in \mathsf{bv}(ts)$.

Relations on type schemes. We say that t' is an instance of $\forall \overline{F}.t$, written $\forall \overline{F}.t \succ t'$, iff there exists ψ with $\mathsf{dom}(\psi) \cap F = \emptyset$ such that $t' = t[\psi]$.

Let $ts = \forall\overline{F}.t$ and $ts' = \forall\overline{F'}.t'$ be type schemes. We then say that ts is more general than ts', written $ts \succeq ts'$, iff $t = t'$ and $F \subseteq F'$. Notice that this is a more "syntactic" (and more restrictive) definition than is standard in the literature; we have chosen the current definition to simplify some proofs later.

Fact 2.2 Suppose $ts \succeq ts'$ and that $ts' \succ t'$. Then also $ts \succ t'$.

Substitution relations. Stating that $\forall\overline{F}.t \succ t'$ amounts to specifying what the bound variables of t (i.e. those not in F) are instantiated to. This motivates the introduction of *substitution relations*, of form $\forall\overline{F}.\vec{g} \succ \vec{g'}$ where one can think of $\vec{g}$ as the bound variables. Such a relation will hold iff there exists ψ with $\mathsf{dom}(\psi) \cap F = \emptyset$ such that $\vec{g'} = \vec{g}[\psi]$. Notice that with $\mathsf{fv}(\vec{g''}) = F$ the relation will hold iff $\forall\overline{\emptyset}.(\vec{g}\vec{g''}) \succ (\vec{g'}\vec{g''})$; hence substitution relations can be represented as semi-unification problems (at the cost of losing some structure). We shall refrain from doing so, however, since semi-unification is undecidable in general and it is unclear if our problem is a general semi-unification problem.

Environments. An environment E is a list of the form $[[x_1 : ts_1] \ldots [x_n : ts_n]]$ (the x_i's not necessarily distinct). We shall often write a type t_i in place of an ts_i: this is a short hand for $\forall\overline{F_i}.t_i$ where $F_i = \mathsf{fv}(t_i)$.

We say that $E(x) = s$ if $s = s_i$ for an i satisfying that $x = x_i$ and that $x \neq x_j$ for all $j > i$ (in other words we look up in E by searching from the right). The result of updating E with $[x : s]$, to be written $E \oplus [x : s]$, is the environment $[[x_1 : s_1] \ldots [x_n : s_n][x : s]]$.

We shall feel free to use the relations and predicates on type schemes also on environments; as an example we say that E is a kernel environments if s_i is a kernel type scheme for all $i \in \{1 \ldots n\}$.

Constraints. There are two kinds of constraints c:

- *C-constraints* (for containment), of form $b_1 \dot{\sqsupseteq} b_2$. This constraint is satisfied iff $b_1 \sqsupseteq b_2$. The result $c[\psi]$ of applying a substitution ψ to this constraint is $b_1[\psi] \dot{\sqsupseteq} b_2[\psi]$;
- *S-constraints* (for substitutions), of form $\forall\overline{F}.\vec{g_1} \dot{\succ} \vec{g_2}$. This constraint is satisfied iff the substitution relation $\forall\overline{F}.\vec{g_1} \succ \vec{g_2}$ holds. The result $c[\psi]$ of applying a substitution ψ to this constraint is $\forall\overline{F'}.\vec{g_1}[\psi] \dot{\succ} \vec{g_2}[\psi]$ where $F' = \mathsf{var}(F[\psi])$.

If $c[\psi]$ is satisfied we say that ψ satisfies c and write $\psi \models c$. Notice that if $\psi \models c$ then also $\psi; \phi \models c$ if c is a C-constraint (by Fact 2.1), but not necessarily if c is an S-constraint – this fact is a major obstacle for solving constraints "sequentially" as we shall see in Section 9. Further notice that $\psi; \phi \models c$ holds iff $\phi \models c[\psi]$.

The capital letter C ranges over lists of constraints and we say that $\psi \models C$ iff $\psi \models c$ holds for all elements c of C. The concatenation of C_1 and C_2 is written as $C_1 \oplus C_2$. The reason for choosing lists rather than sets is that this extra structure makes it possible to formulate some properties of the constraints generated by the reconstruction algorithm – due to space limits this line will not be pursued further in the present paper.

Principal solutions to C-constraints. With the ordering on behaviours axiomatised in Fig. 1 there seems to be no notion of "principal solutions" to C-constraints. For an example of this, consider the constraint $\beta \sqsupseteq \beta; !\mathsf{Int}; !\mathsf{Int}$. The substitution ψ_1 which maps β into $\mathsf{REC}\beta.(\beta; !\mathsf{Int}; !\mathsf{Int})$ as well as the substitution ψ_2 which maps β into $\mathsf{REC}\beta.(\beta; !\mathsf{Int})$ will satisfy this constraint; but using the axiomatisation in Fig. 1 it seems hard to find a sense in which ψ_1 and ψ_2 are comparable.

A remedy, inspired by [CC91] where a similar observation is made concerning recursive types, might be to consider $\mathsf{REC}\beta.b$ equivalent to its infinite unfolding (cf. rule R1 in Fig. 1 which states that $\mathsf{REC}\beta.b$ is equivalent to its finite unfoldings). It is not yet clear whether such a change in the axiomatisation will be sufficient to gain some notion of principality.

Downwards closure. Given a set of variables F and a list of constraints C we define the downwards closure of F with respect to C, written F^C, as the smallest set F' such that $F \subseteq F'$ and such that whenever $\beta \dot{\sqsupseteq} b \in C$ and $\beta \in F'$ then also $\mathsf{fv}(b) \subseteq F'$. It is easy to see that F^C can be computed constructively.

A brief motivation: think of the variables in F as the "non-polymorphic" ones; i.e. those the reconstruction algorithm is not allowed to "make copies of". Then if $\beta \dot{\sqsupseteq} b \in C$ with $\beta \in F$ also the variables in b should be non-polymorphic. We shall elaborate on this in Section 8; more formally we have the following result which is a consequence of Fact 2.1:

Fact 2.3 If $\psi \models C$ then $\mathsf{fv}(F[\psi]) = \mathsf{fv}(F^C[\psi])$.

3 The type and behaviour inference system

In Fig. 2 we list (part of) the inference system for which we are going to present a reconstruction algorithm. A judgement is of the form $E \vdash e : t \,\&\, b$ and says that in (kernel) environment E one can infer that expression e has type t and behaviour b.

For constants a function CTypeOf() is employed, assigning an *extended type scheme* to each constant. An extended type scheme ts is of form $\forall\overline{\emptyset}.(t, C)$ (so $\mathsf{fv}(ts) = \emptyset$). Consistent with previous definitions we say that t' is an instance of $\forall\overline{\emptyset}.(t, C)$, written $\forall\overline{\emptyset}.(t, C) \succ t'$, iff there exists ψ such that $t' = t[\psi]$ and such that $\psi \models C$. The value of CTypeOf() on some constants is tabulated below (adopted from [NN94b, Table 4]).

$$
\begin{array}{lcl}
\text{CTypeOf}(+) & = & \forall\overline{\emptyset}.(\mathsf{Int} \rightarrow^{\beta_1} \mathsf{Int} \rightarrow^{\beta_2} \mathsf{Int}, [\beta_1 \sqsupseteq \epsilon, \beta_2 \sqsupseteq \epsilon]) \\
\text{CTypeOf}(\mathsf{sync}) & = & \forall\overline{\emptyset}.((\alpha\ \mathbf{com}\ \beta_1) \rightarrow^{\beta_2} \alpha, [\beta_2 \sqsupseteq \beta_1]) \\
\text{CTypeOf}(\mathsf{send}) & = & \forall\overline{\emptyset}.(\alpha\ \mathbf{chan} \times \alpha \rightarrow^{\beta_1} \alpha\ \mathbf{com}\ \beta_2, [\beta_1 \sqsupseteq \epsilon, \beta_2 \sqsupseteq !\alpha]) \\
\text{CTypeOf}(\mathsf{receive}) & = & \forall\overline{\emptyset}.(\alpha\ \mathbf{chan} \rightarrow^{\beta_1} \alpha\ \mathbf{com}\ \beta_2, [\beta_1 \sqsupseteq \epsilon, \beta_2 \sqsupseteq ?\alpha]) \\
\text{CTypeOf}(\mathsf{channel}) & = & \forall\overline{\emptyset}.(\mathsf{Unit} \rightarrow^{\beta} \alpha\ \mathbf{chan}, [\beta \sqsupseteq \alpha\ \mathsf{CHAN}]) \\
\text{CTypeOf}(\mathsf{fork}) & = & \forall\overline{\emptyset}.((\mathsf{Unit} \rightarrow^{\beta_1} \alpha) \rightarrow^{\beta_2} \mathsf{Unit}, \beta_2 \sqsupseteq \mathsf{FORK}\ \beta_1)
\end{array}
$$

The inference system is much as in [NN94a], but there are the following differences: (i) we use the non-standard notation for type schemes that was introduced in Section 2; (ii) we have removed regions from behaviours; (iii) we

$$E \vdash x : t \;\&\; b \qquad \text{if } E(x) \succ t \text{ and } b \sqsupseteq \epsilon$$

$$E \vdash c : t \;\&\; b \qquad \text{if CTypeOf}(c) \succ t \text{ and } b \sqsupseteq \epsilon$$

$$\frac{E \vdash e_1 : t_1 \;\&\; b_1,\; E \vdash e_2 : t_2 \;\&\; b_2}{E \vdash e_1\; e_2 : t \;\&\; b} \qquad \text{if } t_1 = t_2 \rightarrow^{b_0} t \text{ and } b \sqsupseteq b_1; b_2; b_0$$

$$\frac{E \oplus [x : t_1] \vdash e_0 : t_0 \;\&\; b_0}{E \vdash \lambda x.e_0 : t \;\&\; b} \qquad \text{if } t = t_1 \rightarrow^{b_0} t_0 \text{ and } b \sqsupseteq \epsilon$$

$$\frac{E \vdash e_1 : t_1 \;\&\; b_1,\; E \oplus [x : \forall \overline{F}.t_1] \vdash e_0 : t \;\&\; b_0}{E \vdash \mathsf{let}\; x{=}e_1 \;\mathbf{in}\; e_0 : t \;\&\; b} \qquad \text{if } b \sqsupseteq b_1; b_0$$
$$\text{where } F = (\mathsf{fv}(E) \cup \mathsf{fv}(b_1)) \cap \mathsf{fv}(t_1)$$

Figure 2: Selected parts of the type and behaviour inference system.

avoid coding the constraints present in the definition of CTypeOf() into the types (as in TypeOf of [NN94a]).

Example 3.1 Consider the following CML-program map2, a version of the well-known map function where a process is forked for each tail while the forking process itself works on the head. A rather informal notation is employed; for instance we write $e_1; e_2$ as an abbreviation for $(\lambda d.e_2)\; e_1$.

```
rec mp2(f) ⇒ λxs.if xs = [] then []
                 else let ch = channel ()
                      in fork (λd.(sync (send ⟨ch,mp2 f (tail xs)⟩)));
                         cons (f (head xs)) (sync (receive ch))
```

We expect to have $\emptyset \vdash \mathsf{map2} : ((\alpha_1 \rightarrow^{\beta_1} \alpha_2) \rightarrow^{\epsilon} (\alpha_1\; \mathbf{list} \rightarrow^{\beta_2} \alpha_2\; \mathbf{list}))\; \&\; \epsilon$ where $\beta_2 = \mathsf{REC}\beta.(\epsilon + ((\alpha_2\; \mathbf{list})\; \mathsf{CHAN}; \mathsf{FORK}\; (\beta; !(\alpha_2\; \mathbf{list})); \beta_1; ?(\alpha_2\; \mathbf{list})))$.

We now list some properties of the inference system (the latter due to Fact 2.2):

Fact 3.2 Suppose $E \vdash e : t \;\&\; b$ and $b' \sqsupseteq b$. Then also $E \vdash e : t \;\&\; b'$.

Fact 3.3 Let E and E' be kernel environments with $E' \succeq E$. Suppose that $E \vdash e : t \;\&\; b$. Then also $E' \vdash e : t \;\&\; b$.

4 A reconstruction algorithm

Our goal is to produce an algorithm which works in the spirit of the well-known algorithm W[Mil78]. Hence we have to perform unification over types; but in order to unify $t_1 \rightarrow^{b} t_2$ with $t_1' \rightarrow^{b'} t_2'$ we have to unify b and b', and if for example $b = b_1; b_2$ and $b' = b_1'; b_2'$ this cannot always be done "compositionally" by first unifying b_1 with b'_1 and then unifying b_2 with b'_2 – for then we will lose the possibility that say $b_1 = b'_2 = \epsilon$ and $b'_1 = b_2 = !\mathsf{Int}$.

The remedy (as done in [TJ92a] and [NN94b] but not in [JG91] where no unification is performed) is to introduce the notion of *simplicity*: a type is simple if all the behaviours it contains are behaviour variables (so $t_1 \rightarrow^b t_2$ is simple iff t_1 and t_2 are both simple and $b = \beta$ for some β); a behaviour is simple if all the types it contains are simple (so e.g. $!t$ is simple iff t is simple) and if it does not contain sub-behaviours of form $\mathsf{REC}\beta.b$; a C-constraint is simple if it is of form $\beta \dot{\sqsupseteq} b$ with b simple; an S-constraint is simple if it is of form $\forall \overline{F}.(\vec{t}\vec{\beta}) \dot{\succ} (\vec{t'}\vec{\beta'})$, with t and t' simple types[1]; a substitution is simple if it maps type variables into simple types and maps behaviour variables into behaviour variables (rather than simple behaviours).

Fact 4.1 Let t be a simple type; let b be a simple behaviour; let C be a list of simple constraints; and let ϕ and ϕ' be simple substitutions. Then $t[\phi]$ is a simple type; $b[\phi]$ is a simple behaviour; $C[\phi]$ is a list of simple constraints; and $\phi; \phi'$ is a simple substitution.

We can now define a procedure UNIFY which takes two simple types t_1 and t_2 and returns the most general unifier if one unifier exists – otherwise UNIFY fails. Some of the more interesting clauses of UNIFY are listed below:

$$
\begin{array}{lll}
\mathsf{UNIFY}(\alpha, t) = \theta & \text{iff} & \theta = [\alpha \mapsto t] \text{ and } (t = \alpha \text{ or } \alpha \notin \mathsf{fv}(t)) \\[1em]
\mathsf{UNIFY}(t_1 \ \mathbf{com}\ \beta_1, t_2 \ \mathbf{com}\ \beta_2) = \theta & \text{iff} & \mathsf{UNIFY}(t_1, t_2) = \theta' \\
 & \text{and} & \beta'_1 = \beta_1[\theta'] \text{ and } \beta'_2 = \beta_2[\theta'] \\
 & \text{and} & \theta = \theta'; [\beta'_1 \mapsto \beta'_2]
\end{array}
$$

Lemma 4.2 Let t_1 and t_2 be simple types. First suppose that $\mathsf{UNIFY}(t_1, t_2) = \theta$. Then θ is simple and $t_1[\theta] = t_2[\theta]$.

Conversely assume that $t_1[\phi] = t_2[\phi]$. Then $\mathsf{UNIFY}(t_1, t_2)$ succeeds with result θ; and there exists ϕ' such that $\phi = \theta; \phi'$.

We are now ready to define the reconstruction algorithm W, as is done in Figure 3. The algorithm fails if and only if a call to UNIFY fails.

The most interesting cases are the ones for identifiers and for polymorphic abstractions:

An identifier x is looked up in the environment and fresh copies $\vec{\gamma'}$ of the bound variables $\vec{\gamma}$ of their type schemes are generated. Moreover, it is recorded (via an S-constraint) that a solution should map $\vec{\gamma'}$ to an instance of what $\vec{\gamma}$ is mapped into in such a way that it does not interfere with the free variables.

For a polymorphic abstraction $\mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0$ the algorithm first finds the type t_1 and behaviour b_1 of e_1; then e_0 is analysed in an environment where x is bound to a type scheme with type part t_1. The free variables of the type scheme (i.e. the non-polymorphic ones) are built from those which are free in the environment together with those which are free in b_1. This corresponds

[1] In fact it holds that t will be of form $\vec{\alpha}$ for all simple S-constraints produced by W.

$$
\begin{array}{ll}
 & W(x, E) = (t, \epsilon, C, id) \\
\text{iff} & E(x) = \forall \overline{F_x}.t_x \\
\text{and} & \vec{\gamma} = \mathsf{fv}(t_x) \setminus F_x \\
\text{and} & \vec{\gamma}' \text{ are fresh copies of } \vec{\gamma} \\
\text{and} & t = t_x[\vec{\gamma} \mapsto \vec{\gamma}'] \\
\text{and} & C = [\forall \overline{F_x}.\vec{\gamma} \,\dot{\succ}\, \vec{\gamma}']
\end{array}
$$

$$
\begin{array}{ll}
 & W(c, E) = (t, \epsilon, C, id) \\
\text{iff} & \text{CTypeOf}(c) = \forall \overline{\emptyset}.(t_c, C_c) \\
\text{and} & \vec{\gamma} = \mathsf{fv}(t_c) \cup \mathsf{fv}(C_c) \\
\text{and} & \vec{\gamma}' \text{ are fresh copies of } \vec{\gamma} \\
\text{and} & t = t_c[\vec{\gamma} \mapsto \vec{\gamma}'] \\
\text{and} & C = C_c[\vec{\gamma} \mapsto \vec{\gamma}']
\end{array}
$$

$$
\begin{array}{ll}
 & W(\lambda x.e_0, E) = (t, \epsilon, C, \theta) \\
\text{iff} & \alpha_1 \text{ and } \beta_0 \text{ are fresh} \\
\text{and} & W(e_0, E \oplus [x : \alpha_1]) \\
 & = (t_0, b_0, C_0, \theta_0) \\
\text{and} & t = \alpha_1[\theta_0] \to^{\beta_0} t_0 \\
\text{and} & C = C_0 \oplus [\beta_0 \dot{\sqsupseteq} b_0] \\
\text{and} & \theta = \theta_0
\end{array}
$$

$$
\begin{array}{ll}
 & W(e_1\ e_2, E) = (t, b, C, \theta) \\
\text{iff} & W(e_1, E) = (t_1, b_1, C_1, \theta_1) \\
\text{and} & W(e_2, E[\theta_1]) = (t_2, b_2, C_2, \theta_2) \\
\text{and} & \alpha \text{ and } \beta_0 \text{ are fresh} \\
\text{and} & \mathsf{UNIFY}(t_1[\theta_2], t_2 \to^{\beta_0} \alpha) = \theta_0 \\
\text{and} & t = \alpha[\theta_0] \\
\text{and} & b = b_1[\theta_2; \theta_0]; b_2[\theta_0]; \beta_0[\theta_0] \\
\text{and} & C = C_1[\theta_2; \theta_0] \oplus C_2[\theta_0] \\
\text{and} & \theta = \theta_1; \theta_2; \theta_0
\end{array}
$$

$$
\begin{array}{ll}
 & W(\mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0, E) = (t, b, C, \theta) \\
\text{iff} & W(e_1, E) = (t_1, b_1, C_1, \theta_1) \\
\text{and} & W(e_0, E[\theta_1] \oplus [x : \forall \overline{\mathcal{NQ}}.t_1]) \\
 & = (t_0, b_0, C_0, \theta_0) \\
\text{and} & t = t_0 \\
\text{and} & b = b_1[\theta_0]; b_0 \\
\text{and} & C = C_1[\theta_0] \oplus C_0 \\
\text{and} & \theta = \theta_1; \theta_0 \\
\multicolumn{2}{l}{\text{where } \mathcal{NQ} = (\mathsf{fv}(E[\theta_1]) \cup \mathsf{fv}(b_1))^{C_1}}
\end{array}
$$

Figure 3: Selected clauses of Algorithm W.

to the inference system in Fig. 2, but notice the following differences: (a) in contrast to the inference system the algorithm does *not* intersect with $\mathsf{fv}(t_1)$ and thus an enriched type scheme is generated; (b) the algorithm takes the downwards closure with respect to the constraints C_1. We shall elaborate upon the former point in Section 6 and upon the latter point in Section 8.

Fact 4.3 Suppose that $W(e, E) = (t, b, C, \theta)$ where the types in E are simple. Then t is a simple type, b is a simple behaviour, C is a list of simple constraints, and θ is a simple substitution.

5 Soundness of algorithm W

We are now going to prove that the algorithm is sound; this means that a solution to the constraints gives rise to a valid inference in the inference system of Figure 2.

Theorem 5.1 Suppose that $W(e, \emptyset) = (t, b, C, \theta)$
and suppose that ψ is such that $\psi \models C$ and $t' = t[\psi]$ and $b' \sqsupseteq b[\psi]$.
Then it holds that $\emptyset \vdash e : t' \;\&\; b'$.

This theorem follows easily from Fact 3.2 and the following proposition that allows a proof by induction:

Proposition 5.2 Suppose that $W(e, E) = (t, b, C, \theta)$. Then for all ψ such that $\psi \models C$ we have $\kappa(E[\theta][\psi]) \vdash e : t[\psi] \,\&\, b[\psi]$. (Recall that W deals with enriched environments whereas the inference system of Figure 2 deals with kernel environments.)

Proof: We perform induction on e; to conserve space we use the same terminology as in the definition of the relevant clause for W. There are seven cases; here we shall list the two most interesting ones only.

The case $W(x, E)$**:** Let $F' = \mathsf{var}(F_x[\psi])$ and let $F'_x = F' \cap \mathsf{fv}(t_x[\psi])$.

Our aim is to show that $\kappa(E[\psi])(x) \succ t[\psi]$, which amounts to

$$\forall \overline{F'_x}.(t_x[\psi]) \succ t_x[\vec{\gamma} \mapsto \vec{\gamma'}][\psi]. \tag{1}$$

Since $\psi \models C$ we have $\forall \overline{F'}.\vec{\gamma}[\psi] \succ \vec{\gamma'}[\psi]$; so there exists a ϕ' with $\mathsf{dom}(\phi') \cap F' = \emptyset$ such that $\psi; \phi' \stackrel{\vec{\gamma}}{=} [\vec{\gamma} \mapsto \vec{\gamma'}]; \psi$. This implies, since $\mathsf{var}(t_x) \subseteq \vec{\gamma} \cup F_x$, that we even have $\psi; \phi' \stackrel{\mathsf{var}(t_x)}{=} [\vec{\gamma} \mapsto \vec{\gamma'}]; \psi$. But this shows that (1) holds, with ϕ' as the "instance substitution".

The case $W(\mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0, E)$**:** First a *remark:* the only property of $\mathcal{NQ}$ we shall exploit is the following which is a consequence of Fact 2.3 (the condition may seem superfluous but is added to make (2) and (5) dual):

$$\psi \models C_1 \text{ implies } \mathsf{var}(\mathcal{NQ}[\psi]) \supseteq \mathsf{var}(\mathsf{fv}(E[\theta_1])[\psi]) \cup \mathsf{var}(b_1[\psi]). \tag{2}$$

First observe that $\theta_0; \psi \models C_1$; so we can apply the induction hypothesis on the call $W(e_1, E)$ and the substitution $\theta_0; \psi$ to get

$$\kappa(E[\theta][\psi]) \vdash e_1 : t_1[\theta_0; \psi] \,\&\, b_1[\theta_0; \psi].$$

Since $\psi \models C_0$ we can apply the induction hypothesis on the call $W(e_0, E[\theta_1] \oplus [x : \ldots])$ and the substitution ψ to get

$$\kappa(E[\theta][\psi]) \oplus [x : \forall \overline{F'}.(t_1[\theta_0; \psi])] \vdash e_0 : t_0[\psi] \,\&\, b_0[\psi]$$

where $F' = \mathsf{fv}(\mathcal{NQ}[\theta_0; \psi]) \cap \mathsf{fv}(t_1[\theta_0; \psi])$.

Let $F'' = (\mathsf{fv}(\kappa(E[\theta][\psi])) \cup \mathsf{fv}(b_1[\theta_0; \psi])) \cap \mathsf{fv}(t_1[\theta_0; \psi])$. Applying (2) on the substitution $\theta_0; \psi$ tells us that $F'' \subseteq F'$ so by Fact 3.3 we infer that

$$\kappa(E[\theta][\psi]) \oplus [x : \forall \overline{F''}.t_1[\theta_0; \psi]] \vdash e_0 : t_0[\psi] \,\&\, b_0[\psi].$$

This shows that we have the desired judgment

$$\kappa(E[\theta][\psi]) \vdash \mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0 : t_0[\psi] \,\&\, b[\psi].$$

6 The need for enriched type schemes

Enriched type schemes in algorithm W. The reconstruction algorithm from [NN94b] carries constraints around, together with type schemes in kernel form. Our algorithm carries no constraints in the environment, but in order to preserve soundness the type schemes have to be enriched. This is informally illustrated by the following example.

Let $e = \mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0$ and suppose that $W(e_1, E) = (t_1, b_1, C_1, \theta_1)$ where $t_1 = \mathsf{Unit} \rightarrow^{\beta} \mathsf{Unit}$ and where C_1 contains the constraint $\beta \dot{\sqsupseteq} !\alpha$, where $\alpha \in \mathsf{fv}(E)$. The intuitive interpretation of this is that when analysing e_0, β is a polymorphic variable with a non-polymorphic subpart α. Thus β cannot be instantiated freely; hence the need to carry α around in the environment (via the type scheme $\forall\overline{\{\alpha\}}.t_1$).

Enriched type schemes in the completeness proof. It turns out to be hard to give a direct proof of the completeness of W with respect to the inference system of Fig. 2, due to the fact that for inferences in this system it does not necessarily hold that "bound variables are unique" (space limits prevent us from giving a more precise definition and/or an example of this phenomenon). We shall alleviate this by presenting an alternative (but equivalent) inference system having the aforementioned property:

An enriched inference system. Inferences are of form $E \vdash_2 e : t \mathbin{\&} b$, where E is an enriched environment. The system is as in Fig. 2, except that the rule for polymorphic abstraction is as follows:

$$\frac{\begin{array}{c} E \vdash_2 e_1 : t_1 \mathbin{\&} b_1 \\ E \oplus [x : \forall\overline{F}.t_1] \vdash_2 e_0 : t \mathbin{\&} b_0 \end{array}}{E \vdash_2 \mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0 : t \mathbin{\&} b} \text{ if } F = \mathsf{fv}(E) \cup \mathsf{fv}(b_1) \text{ and } b \sqsupseteq b_1; b_0. \qquad (3)$$

The two inference systems are equivalent in the following sense:

Theorem 6.1 $\emptyset \vdash e : t \mathbin{\&} b$ holds iff $\emptyset \vdash_2 e : t \mathbin{\&} b$ holds.

This is a consequence of the following two propositions:

Proposition 6.2 Suppose $E \vdash_2 e : t \mathbin{\&} b$. Then also $\kappa(E) \vdash e : t \mathbin{\&} b$.

Proposition 6.3 Suppose $E \vdash e : t \mathbin{\&} b$ and that $\kappa(E') = E$. Then also $E' \vdash_2 e : t \mathbin{\&} b$.

7 Completeness of algorithm W

We are now in a position to prove that algorithm W is complete: if there exists a valid inference then the algorithm will produce a set of constraints that can be satisfied.

Theorem 7.1 Suppose $\emptyset \vdash e : t' \mathbin{\&} b'$. Then $W(e, \emptyset) = (t, b, C, \theta)$ and there exists a ψ such that $\psi \models C$ and $t' = t[\psi]$ and $b' \sqsupseteq b[\psi]$.

Notice the symmetry with Theorem 5.1. The theorem follows easily from Theorem 6.1 and the following proposition that admits a proof by induction:

Proposition 7.2 Suppose $E' \vdash_2 e : t' \ \& \ b'$ and $E[\phi] \succeq E'$. Then $W(e, E) = (t, b, C, \theta)$ and there exists a ψ such that $\theta; \psi \stackrel{\mathsf{var}(E)}{=} \phi$ and $\psi \models C$ and $t[\psi] = t'$ and $b' \sqsupseteq b[\psi]$.

Proof: Induction on the derivation tree for $E' \vdash_2 e : t' \ \& \ b'$; to conserve space we use the same terminology as in the definition of the relevant clause for W and we write e.g. $\psi \stackrel{E}{=} \phi$ instead of $\psi \stackrel{F}{=} \phi$ where $F = \mathsf{var}(E)$. There are seven cases; here we shall list the two most interesting ones only.

The case $e = x$: Suppose $E' \vdash_2 x : t' \ \& \ b'$ holds because $E'(x) = \forall \overline{F'_x}.t'_x$, because $t' = t'_x[\phi']$ with $\mathsf{dom}(\phi') \cap F'_x = \emptyset$, and because $b' \sqsupseteq \epsilon$.

Since $E[\phi] \succeq E'$ it holds that $t_x[\phi] = t'_x$ (thanks to our syntactic definition of $\succeq$) and that $\mathsf{var}(F_x[\phi]) \subseteq F'_x$. From this we infer that

$$\mathsf{dom}(\phi') \cap \mathsf{var}(F_x[\phi]) = \emptyset. \tag{4}$$

Now define ψ as follows: it maps $\vec{\gamma'}$ into $\vec{\gamma}[\phi;\phi']$; and otherwise it behaves like ϕ. This ensures that $\theta; \psi \stackrel{E}{=} \phi$; and it is trivial that $b' \sqsupseteq b[\psi]$.

For our remaining claims, observe that due to (4) we have

$$[\vec{\gamma} \mapsto \vec{\gamma'}]; \psi \stackrel{\vec{\gamma} \cup F_x}{=} \psi; \phi' \text{ with } \mathsf{dom}(\phi') \cap \mathsf{fv}(F_x[\psi]) = \emptyset$$

showing that $\forall \overline{F_x[\psi]}.\vec{\gamma}[\psi] \ \succ \ \vec{\gamma'}[\psi]$ implying $\psi \models C$; and that (since $\mathsf{var}(t_x) \subseteq F_x \cup \vec{\gamma}$) $t[\psi] = t_x[[\vec{\gamma} \mapsto \vec{\gamma'}]; \psi] = t_x[\psi;\phi'] = t_x[\phi;\phi'] = t'_x[\phi'] = t'$.

The case $e = \mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0$: First a *remark:* the only property of $\mathcal{NQ}$ we shall exploit is the following:

$$\psi \models C_1 \text{ implies } \mathsf{var}(\mathcal{NQ}[\psi]) \subseteq \mathsf{var}(\mathsf{fv}(E[\theta_1])[\psi]) \cup \mathsf{var}(b_1[\psi]). \tag{5}$$

Notice the duality between (2) and (5); and notice that the conjunction of these two is a consequence of Fact 2.3.

Some terminology: we define $V_1 = \mathsf{var}(t_1, b_1, C_1, \theta_1)$ and $V_0 = \mathsf{var}(t_0, b_0, C_0, \theta_0)$ and $V_e = \mathsf{var}(E[\theta_1]) \cup \mathcal{NQ} \cup \mathsf{fv}(t_1)$; and say that a variable is "internal" if it occurs in V_1 but not in V_e. We can assume that no internal variable occurs in V_0.

Suppose $E' \vdash_2 \mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0 : t' \ \& \ b'$ because of $E' \vdash_2 e_1 : t'_1 \ \& \ b'_1$ and of $F' = \mathsf{fv}(E') \cup \mathsf{fv}(b'_1)$ and of $E' \oplus [x : \forall \overline{F'}.t'_1] \ \vdash_2 e_0 : t' \ \& \ b'_0$ and because of $b' \sqsupseteq b'_1; b'_0$. By induction we see that $W(e_1, E)$ succeeds and that there exists ψ_1 such that $\theta_1; \psi_1 \stackrel{E}{=} \phi$, such that $\psi_1 \models C_1$, such that $t_1[\psi_1] = t'_1$, and such that $b'_1 \sqsupseteq b_1[\psi_1]$. It holds that

$$\mathsf{var}(\mathcal{NQ}[\psi_1]) \subseteq F'$$

for by (5) we have $\mathsf{var}(\mathcal{NQ}[\psi_1]) \subseteq \mathsf{var}(\mathsf{fv}(E[\theta_1])[\psi_1]) \cup \mathsf{var}(b_1[\psi_1])$, by assumption we have $\mathsf{var}(\mathsf{fv}(E[\theta_1])[\psi_1]) = \mathsf{fv}(E[\theta_1][\psi_1]) = \mathsf{fv}(E[\phi]) \subseteq \mathsf{fv}(E')$, and by Fact 2.1 we have $\mathsf{fv}(b_1[\psi_1]) \subseteq \mathsf{fv}(b'_1)$.[2]

[2]It is not possible to replace this subset inclusion by set equality. This is the motivation for making $E[\phi] \succeq E'$ (and not $E[\phi] = E'$) be part of the induction hypothesis.

Using the above observation it is straightforward to verify that

$$(E[\theta_1] \oplus [x : \forall\overline{\mathcal{NQ}}.t_1])[\psi_1] \succeq E' \oplus [x : \forall\overline{F'}.t_1']$$

so by induction we see that $W(e_0, \_)$ succeeds and that there exists ψ_0 such that $\theta_0; \psi_0 \overset{V_e}{=} \psi_1$, such that $\psi_0 \models C_0$, such that $t_0[\psi_0] = t'$ and such that $b_0' \sqsupseteq b_0[\psi_0]$.

Now define ψ to behave as ψ_0 except that is behaves as ψ_1 on internal variables. We have the following relations:

$$\psi \overset{V_0}{=} \psi_0 \text{ and } \theta_0; \psi \overset{V_1 \cup V_e}{=} \psi_1 \tag{6}$$

where the second part follows from the following reasoning: if γ is internal then $\gamma[\theta_0; \psi] = \gamma[\psi] = \gamma[\psi_1]$; and if γ is not internal (and hence belongs to V_e) then $\gamma[\theta_0]$ does not contain any internal variables so $\gamma[\theta_0; \psi] = \gamma[\theta_0; \psi_0] = \gamma[\psi_1]$.

Using (6) enables us to infer the desired properties of ψ: (i) if $\gamma \in \mathsf{var}(E)$ then $\gamma[\theta; \psi] = \gamma[\theta_1][\theta_0; \psi] = \gamma[\theta_1][\psi_1] = \gamma[\phi]$; (ii) $\psi \models C$ holds because $\psi \models C_1[\theta_0]$ (which follows from $\psi_1 \models C_1$) and because $C_0[\psi] = C_0[\psi_0]$; (iii) $t[\psi] = t_0[\psi_0] = t'$; (iv) we infer that $b' \sqsupseteq b_1'; b_0' \sqsupseteq b_1[\psi_1]; b_0[\psi_0] = b_1[\theta_0][\psi]; b_0[\psi] = b[\psi]$.

8 Choice of generalisation strategy

In this section we shall discuss alternative ways of defining the entity $\mathcal{NQ}$ occurring in Figure 3 (the clause for polymorphic abstractions). Recall that $\mathcal{NQ}$ satisfies the following property:

$$\beta \dot{\sqsupseteq} b \in C_1 \text{ and } \beta \in \mathcal{NQ} \text{ implies } \mathsf{fv}(b) \subseteq \mathcal{NQ}$$

stating that if a variable is not polymorphic then neither of its "subparts" can be polymorphic. We now consider two variations on this theme.

8.1 An algorithm not taking the downwards closure

First define a version of W, to be called W^n for <u>n</u>o closure, where simply $\mathcal{NQ} = \mathsf{fv}(E[\theta_1]) \cup \mathsf{fv}(b_1)$. It is trivial that (2) and (5) still hold and hence W^n will be sound as well as complete. On the other hand, failures are reported at a later stage as witnessed by the expression $e = \mathsf{let}\ ch{=}\mathsf{channel}\ ()\ \mathbf{in}\ e_1$ where in e_1 an integer as well as a boolean is transmitted over the newly generated channel ch. W^n applied to e will terminate successfully and return constraints including the following

$$[\beta \dot{\sqsupseteq} \alpha\ \mathsf{CHAN},\ \forall\overline{\{\beta\}}.\alpha \dot{\succ} \mathsf{Bool},\ \forall\overline{\{\beta\}}.\alpha \dot{\succ} \mathsf{Int}]$$

which are easily seen to be unsolvable; whereas W applied to e will fail immediately (since α is considered a non-polymorphic variable and hence is not copied, causing UNIFY to fail). So it seems that W should be preferred to W^n on the basis that failures should be reported as early as possible.

Note that e above can be typed if ch is assigned the type scheme $\forall\overline{\emptyset}.\alpha$ **chan** but cannot be typed if ch is assigned the type scheme $\forall\overline{\{\alpha\}}.\alpha$ **chan**. The former case will arise if α **chan** is handled as α **list**; whereas the latter case will arise if α **chan** is handled by the techniques for α **ref** developed in [TJ92b] and [BD93].

8.2 An algorithm taking the upwards closure

Next define a version of W, to be called W^{ud} for <u>u</u>pwards and <u>d</u>ownwards closure, where $\mathcal{NQ}$ is defined in such a way that

$$\beta \dot{\sqsupseteq} b \in C_1 \text{ and } (\mathsf{fv}(b) \cup \{\beta\}) \cap \mathcal{NQ} \neq \emptyset \text{ implies } (\mathsf{fv}(b) \cup \{\beta\}) \subseteq \mathcal{NQ}$$

which in effect states that in order for a variable to be polymorphic, all of its "subparts" should be polymorphic as well. As we shall see this approach to separating polymorphic and non-polymorphic variables is too strong. On the positive side (2) still holds so W^{ud} is sound. On the negative side (5) does not hold; in fact completeness fails because we have found a well-typed expression upon which W^{ud} does not succeed.

The algorithm presented in [NN94b] essentially works similar to W^{ud}.

9 Solving the constraints

In this section we discuss how to solve the constraints generated by Algorithm W. As stated in Fact 4.3, the C-constraints are of form $\beta \dot{\sqsupseteq} b$ and the S-constraints are of form $\forall \overline{F}.\vec{t}\vec{\beta} \dot{\succ} \vec{t'}\vec{\beta'}$ (where actually, among other things, $\vec{t}$ is of form $\vec{\alpha}$ and where $\vec{\alpha}$ as well as $\vec{\beta}$ are disjoint variables). The right hand sides of the C-constraints may be quite lengthy, for instance they will often involve sub-behaviours of form $\epsilon; \epsilon; \ldots$, but we have implemented an algorithm that applies the behaviour equivalences from Fig. 1 so as to decrease the size of behaviours significantly (in most cases).

Solving constraints sequentially. Given a set of constraints C, a natural way to search for a substitution ψ that satisfies C is to proceed sequentially:

- if C is empty, let $\psi = id$;
- otherwise, let C be the disjoint union of C' and C''. Suppose we can find ψ' which solves C'. Next suppose we (recursively) can find ψ'' which solves $C''[\psi']$. Then return $\psi = \psi'; \psi''$.

It is easy to see that $\psi \models C$ provided $C'[\psi']$ is such that for all ϕ we have $\phi \models C'[\psi']$. This will be the case if $C'[\psi]$ only contains C-constraints (due to Fact 2.1) and S-constraints of form $\forall \overline{F}.\vec{g} \dot{\succ} \vec{g}$, the latter kind to be denoted S-equalities. So we arrive at the following (informally specified) condition:

$$\text{S-constraints are not solved unless they become S-equalities.} \tag{7}$$

Moreover, we have the following (also informally specified) condition:

$$\text{applying } \psi' \text{ to } C'' \text{ should preserve any "nice structure" that } C'' \text{ has.} \tag{8}$$

Some constraints we know how to solve. If C is a list of constraints, such that all S-constraints in C are of form $\forall \overline{F}.(\vec{\alpha}, \vec{\beta}) \dot{\succ} (\vec{\alpha'}, \vec{\beta'})$, we can apply the scheme for sequential solution outlined in the preceding section (we shall not deal with other kinds of S-constraints even though some of those might have simple solutions as well).

First we convert the S-constraints into S-equalities, cf. (7). This is done by identifying all type and behaviour variables occurring in "corresponding positions" (thus going from a polymorphic world into a monomorphic world).

Next we have to solve the C-constraints sequentially; and the invariant to be preserved during this process (cf. (8)) is that they should be of form $\beta \dot{\sqsupseteq} b$ (where b is no longer assured to be a simple behaviour, as it may contain recursion). It is easy to see that this invariant will be maintained provided we can solve a constraint set of the form $\{\beta \dot{\sqsupseteq} b_1, \ldots, \beta \dot{\sqsupseteq} b_n\}$ by means of a substitution whose domain is $\{\beta\}$. But this can easily be achieved, due to rule R1 in Figure 1: just map β into $\mathsf{REC}\beta.b_1 + \ldots + b_n$ (if β does not occur in the b_i's, we can omit the recursion). This is also the way that C-constraints are solved in [NN94b].

10 Conclusion

In this paper we have adapted the traditional algorithm W to our type and behaviour system. The algorithm has been implemented and has provided quite illuminating analyses of example CML programs.

One difference from the traditional formulation of W is that we generate so-called C-constraints that then have to be solved. This is a consequence of our behaviours being a non-free algebra and is an phenomenon found also in [JG91].

Another and major difference from the traditional formulation, as well as that of [JG91], is that we generate so-called S-constraints that also have to be solved. This phenomenon is needed because our C-constraints would seem not to have principal solutions (as was discussed in Section 2). This in not the case for the traditional "free" unification of Standard ML, but it is a phenomenon well-known in unification theory [Sie89]. As a consequence we have to ensure that the different solutions to the C-constraints (concerning the polymorphic definition and its instantiations) are comparable and this is the purpose of the S-constraints. At present it is an open problem how hard it is to solve S-constraints in the presence of C-constraints and whether it is as hard as general semi-unification (which is undecidable). This problem is closely related to the question of whether all expressions typeable in (C)ML are also typable in our system. More precisely consider an expression e that is typeable in Standard ML when types α **chan** are handled by the techniques for α **ref** developed in [TJ92b]; the (open) question then is whether this expression is always typeable in our system. If this should ever fail it means that the causality espressed in the behaviours constitute an unwarranted restriction upon the language.

Future work will show how to overcome these problems. One avenue of research is to extend our axiomatization of $\sqsupseteq$ such that principal solutions exist; perhaps by identifying recursive behaviours with their infinite unfolding just as done in [CC91] for recursive types. Another avenue is to accept that there definitely will be applications of "effect systems" in theories that are known not to have principal solutions; in this case we may try to reduce our problems by incorporating a more general rule for subtyping into the inference system.

Acknowledgements This research has been supported by the DART and LOMAPS projects.

References

[BD93] Dominique Bolignano and Mourad Debabi. A coherent type system for a concurrent, functional and imperative programming language. In *AMAST '93*, 1993.

[BS94] Bernard Berthomieu and Thierry Le Sergent. Programming with behaviours in an ML framework: the syntax and semantics of LCS. In *ESOP '94*, volume 788 of *LNCS*, pages 89–104. Springer-Verlag, 1994.

[CC91] Felice Cardone and Mario Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92:48–80, 1991.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[JG91] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *ACM Symposium on Principles of Programming Languages*, pages 303–310. ACM Press, 1991.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[NN93] Flemming Nielson and Hanne Riis Nielson. From CML to process algebras. In *CONCUR '93*, volume 715 of *LNCS*. Springer-Verlag, 1993. An expanded version appears as DAIMI technical report no. PB-433.

[NN94a] Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology. In *ACM Symposium on Principles of Programming Languages*. ACM Press, January 1994.

[NN94b] Flemming Nielson and Hanne Riis Nielson. Constraints for polymorphic behaviours of concurrent ML. In *Constraints in Computational Logics (CCL '94)*, volume 845 of *LNCS*. Springer-Verlag, September 1994.

[NN94c] Hanne Riis Nielson and Flemming Nielson. Static and dynamic processor allocation for higher-order concurrent languages. Technical Report PB-483, DAIMI, University of Aarhus, Denmark, 1994. Also see this volume.

[PGM90] Sanjiva Prasad, Alessandro Giacalone, and Prateek Mishra. Operational and algebraic semantics for Facile: A symmetric integration of concurrent and functional programming. In *ICALP 90*, 1990.

[Rep91] John H. Reppy. CML: A higher-order concurrent language. In *SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991.

[Sie89] Jörg H. Siekmann. Unification theory. *J. Symbolic Computation*, 7:207–274, 1989.

[TJ92a] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programing*, 2(3):245–271, 1992.

[TJ92b] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *IEEE Symposium on Logic in Computer Science*, pages 162–173, 1992. Also see Information and Computation **111** *2*, 1994.