J. Functional Programming **1** (1): 1–000, January 1993

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 representation 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 take place if the function is applied or the delayed action is activated. The development of the present paper improves a previously published algorithm in achieving completeness as well as soundness; this is due to an alternative strategy for generalising over types and behaviours.

1 Introduction

It is well-known that testing can only demonstrate the presence of errors, 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 developed, because they allow static checks of certain kinds of errors: 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 (Reppy, 1991) that extends Standard ML with concurrency, Facile (Prasad *et al.*, 1990) that follows a similar approach but more directly contains syntax for expressing CCS-like process composition, and LCS (Berthomieu and Sergent, 1994). The overall communication structure of such programs may not be immediately clear to the

programmer 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 (Nielson and Nielson, 1993) and (Nielson and Nielson, 1994a) inference systems are developed that extend the usual notion of types with behaviours. Applications of such information are demonstrated in (Nielson and Nielson, 1994a) and (Nielson and Nielson, 1995).

The question remains: how to implement the inference system, i.e. how to reconstruct the types and behaviours? It seems appropriate to use a modified version of algorithm W (Milner, 1978). 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 (Jouvelot and Gifford, 1991; Talpin and Jouvelot, 1992); in this approach 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 (Nielson and Nielson, 1994b), 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. The use of S-constraints is *not* a standard tool for the analysis of polymorphic languages; however, they seem to be needed because the C-constraints apparently lack a "principal solution property" (a phenomenon well-known in unification theory). Finding a "canonical" solution to C-constraints may be done as in (Nielson and Nielson, 1994b); in sufficiently simple cases this solution can be shown to be "principal".

The present paper improves on (Nielson and Nielson, 1994b) by (i) achieving completeness in addition to soundness (by means of another generalisation strategy and another formulation of S-constraints), and (ii) avoiding some redundancy in the generated constraints. For simple cases we show how to solve the constraints generated, but it remains an open problem how to solve the constraints in general and how to characterise the solution as "principal".

Overview of paper

Section 2 and 3 set up the background for the present work: in Section 2 we give a brief introduction to CML and behaviours, and in Section 3 we present the inference system from (Nielson and Nielson, 1994a). Section 4 contains a detailed motivation for our design of the reconstruction algorithm W. In Section 5 and 6 the algorithm is shown to be sound and complete. Section 7 elaborates on our choice of generalisation strategy. In Section 8 we show how to solve the constraints generated in special cases. Section 9 concludes, and example output from our prototype implementation is shown in Appendix A.

2 CML-expressions and behaviours

CML-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 $\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$, recursive function definitions $\mathsf{rec}\ f(x) \Rightarrow e_0$, and sequential composition $e_1;e_2$. This is much like ML, the concurrent aspects being taken care of by the constants c some of which will appear in the example below.

Example 2.1

The following CML-program map2 is a version of the well-known map function except that a process is forked for each tail while the forking process itself works on the head. A channel over which the communication takes place is allocated by means of channel; then fork creates a new process which computes map2 f (tail xs) and sends the result over the channel. The purpose of the constant sync is to convert a communication *possibility* into an actual communication (see (Reppy, 1991) for further motivation).

```
rec map2(f) ⇒ λxs.if xs = [] then []
                  else let ch = channel ()
                       in fork (λd.(sync (send ⟨ch,map2 f (tail xs)⟩)));
                          cons (f (head xs)) (sync (receive ch))
```

The "underlying type" of map2 will be $(\alpha_1 \to \alpha_2) \to (\alpha_1\ \mathbf{list} \to \alpha_2\ \mathbf{list})$ and annotation with *behaviour* information yields the type

$$(\alpha_1 \to^{\beta_1} \alpha_2) \to^{\epsilon} (\alpha_1\ \mathbf{list} \to^{b_2} \alpha_2\ \mathbf{list})$$

where b_2 (the behaviour of map2 f) is expressed in terms of β_1 (the behaviour of f) as follows:

$$\mathsf{REC}\beta.(\epsilon + ((\alpha_2\ \mathbf{list})\,\mathsf{CHAN};\ \mathsf{FORK}\,(\beta;\ !(\alpha_2\ \mathbf{list}));\ \beta_1;\ ?(\alpha_2\ \mathbf{list}))).$$

As we cannot statically predict which branch is taken we need the choice operator "$+$": if the list xs is empty then b_2 performs no concurrent actions (this is what ϵ denotes). Otherwise it will first allocate a channel which transmits values of type $\alpha_2\ \mathbf{list}$; then it forks a process which first calls b_2 recursively (to work on the tail of the list) and then outputs a value of type $\alpha_2\ \mathbf{list}$; then it performs β_1 (by computing f on the head of the list); and finally it receives a value of type $\alpha_2\ \mathbf{list}$. □

The above example demonstrates that the use of behaviours enables us to express the essential communication properties of a CML program in a compact way and thus supports a two-stage approach to program analysis: instead of writing a number of analyses for CML programs one writes these analyses for behaviours (presumably a much easier task) and then relies on *one* analysis mapping CML programs into behaviours. The semantic soundness of this approach follows from the subject reduction theorem from (Nielson and Nielson, 1994a).

Some useful analyses on behaviours. In (Nielson and Nielson, 1994a) a behaviour is tested for whether it has finite communication topology, that is whether only finitely many processes are spawned and whether only finitely many channels are created. If the former is the case we may dispense with multitasking; if the latter is the case we may dispense with multiplexing. Both cases may lead to substantial savings in the run-time system. In (Nielson and Nielson, 1995) two analyses are presented which provide information helpful for making a static (resp. dynamic) decision about where to allocate processes.

Types. Types t can be either a type variable α, a base type like int or bool or unit, a product type $t_1 \times t_2$, 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 is 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).

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 concurrent action takes 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). A recursive behaviour $b = \mathsf{REC}\beta.\,b'$ binds β in the sense that the set of free variables $\mathsf{fv}(b)$ is defined to be $\mathsf{fv}(b') \setminus \{\beta\}$; and we assume alpha-conversion to be performed freely.

Compared to (Nielson and Nielson, 1994a) we have omitted *regions* as these present no additional problems to the algorithm.

3 The type and behaviour inference system

In Fig. 1 (explained below) we list the inference system. A judgement is of the form $E \vdash e : t \;\&\; b$ and says that in the environment E one can infer that expression e has type t and behaviour b. An environment is a list of type schemes where the result of updating E with $[x : ts]$ is written $E \oplus [x : ts]$; as usual type schemes take the form $\forall\vec{\gamma}.t$ where γ ranges over type variables and behaviour variables collectively and as usual we identify a type t with the type scheme $\forall\emptyset.t$.

As is always the case for program analysis we shall be interested in getting as precise information as possible, but due to decidability issues approximations are needed. We shall approximate behaviours but not types, that is we have "subeffecting" (cf. (Tang, 1994)) but not "subtyping". To formalise this we impose a preorder $\sqsupseteq$

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

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

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

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

$$\frac{E \vdash e_1 : t_1 \;\&\; b_1,\ E \oplus [x : \mathsf{gen}(t_1, E, b_1)] \vdash e_0 : t \;\&\; b_0}{E \vdash \mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0 : t \;\&\; b} \qquad \text{if } b \sqsupseteq b_1; b_0$$

$$\frac{E \vdash e_0 : \mathsf{bool} \;\&\; b_0,\ E \vdash e_1 : t \;\&\; b_1,\ E \vdash e_2 : t \;\&\; b_2}{E \vdash \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : t \;\&\; b} \qquad \text{if } b \sqsupseteq b_0; (b_1 + b_2)$$

$$\frac{E \oplus [f : t \rightarrow^{b_0} t'] \oplus [x : t] \vdash e_0 : t' \;\&\; b_0}{E \vdash \mathsf{rec}\ f(x) \Rightarrow e_0 : t \rightarrow^{b_0} t' \;\&\; b} \qquad \text{if } b \sqsupseteq \epsilon$$

$$\frac{E \vdash e_1 : t_1 \;\&\; b_1,\ E \vdash e_2 : t \;\&\; b_2}{E \vdash e_1; e_2 : t \;\&\; b} \qquad \text{if } b \sqsupseteq b_1; b_2$$

Fig. 1. The type and behaviour inference system.

on behaviours just as in (Nielson and Nielson, 1994a, Table 3), with the intuitive interpretation that if $b \sqsupseteq b'$ then b approximates b' in the sense that any action performed by b' can also be performed by b. (To be more precise: $\sqsupseteq$ is a subset of the simulation ordering which is undecidable, whereas $\sqsupseteq$ is decidable for behaviours not containing recursion (Nielson and Nielson, 1996).) This approximation is "inlined" into all the clauses of the inference system and yields:

Fact 3.1

If $E \vdash e : t \;\&\; b$ and $b' \sqsupseteq b$ then $E \vdash e : t \;\&\; b'$. □

The preorder is axiomatised in Figure 2, where $b_1 \equiv b_2$ denotes that $b_1 \sqsupseteq b_2$ and $b_2 \sqsupseteq b_1$ (whereas "=" denotes syntactic equality) and where $b[\phi]$ denotes the result of applying the substitution ϕ to b. The axiomatisation expresses that "; " is associative (S1) with ϵ as neutral element (E1,E2); that "$\sqsupseteq$" is a congruence wrt. the various constructors (C1,C2,C3,C4); and that $+$ is least upper bound wrt. $\sqsupseteq$ (J1,J2 together with C2 and P2).

Fact 3.2

If $b_1 \sqsupseteq b_2$ then $\mathsf{fv}(b_1) \supseteq \mathsf{fv}(b_2)$ and $b_1[\phi] \sqsupseteq b_2[\phi]$. □

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

Fig. 2. The preorder $\sqsupseteq$ with equivalence $\equiv$.

We now return to Fig. 1. The clause for monomorphic abstraction says that if e_0 in an environment where x is bound to t_1 has type t_0 and behaviour b_0, then $\lambda x.e_0$ has type $t_1 \rightarrow^{b_0} t_0$. The clause for application reflects the call-by-value semantics of CML: first the function part is evaluated (b_1); then the argument part is evaluated (b_2); finally the function is called on the argument (b_0). The clause for an identifier x says that its type t must be a polymorphic instance of the type scheme $E(x)$ whereas the behaviour is ϵ (again reflecting that CML is call-by-value); as usual $\forall\vec{\gamma}.t \succ t'$ denotes that there exists ϕ with $\mathsf{dom}(\phi) \subseteq \vec{\gamma}$ such that $t' = t[\phi]$. The clause for a polymorphic abstraction $\mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0$ reflects that first e_1 is evaluated (exactly once) and then e_0 is evaluated; the polymorphic aspects are taken care of by the function $\mathsf{gen}(t_1, E, b_1)$:

$$\mathsf{gen}(t_1, E, b_1) = \forall\vec{\gamma}.t_1 \text{ where } \vec{\gamma} = \mathsf{fv}(t_1) \setminus (\mathsf{fv}(E) \cup \mathsf{fv}(b_1)) \tag{1}$$

It generalises (we shall also use the term "quantifies") all variables in t_1 except those which are free in the environment E (which is a standard requirement) and except those which are free in the behaviour b_1 (which is a standard requirement for effect systems (Talpin and Jouvelot, 1994)). The clause for sequential compositions $e_1;e_2$ reflects that first e_1 is evaluated (for its side effects) and then e_2 is evaluated to produce a value (and some side effects).

For a constant c the type t must be a polymorphic instance of CTypeOf(c) which is an *extended type scheme*, that is of form $\forall\vec{\gamma}.(t, C)$ where C is a set of constraints of form $b_1 \sqsupseteq b_2$ — such constraints are denoted *C-constraints* (for containment), and we say that a substitution ϕ satisfies C if for all $(b_1 \sqsupseteq b_2) \in C$ it holds that $b_1[\phi] \sqsupseteq b_2[\phi]$ (according to Fig. 2). Now $(\forall\vec{\gamma}.(t, C)) \succ t'$ holds iff there exists ϕ with $\mathsf{dom}(\phi) \subseteq \vec{\gamma}$ such that $t' = t[\phi]$ and such that ϕ satisfies C. In addition we shall demand each CTypeOf(c) to be closed, that is $\vec{\gamma} = \mathsf{fv}(t, C)$; accordingly we shall allow to write $\forall(t, C)$ for $\forall\vec{\gamma}.(t, C)$ where $\vec{\gamma} = \mathsf{fv}(t, C)$. Below we tabulate the value of CTypeOf() on some constants (all occurring in Example 2.1), this is adopted from (Nielson and Nielson, 1994b, Table 4).

$$
\begin{array}{ll}
\mathsf{head}: & \forall(\alpha\ \mathbf{list} \rightarrow^{\beta} \alpha, [\beta \sqsupseteq \epsilon\,]) \\
\mathsf{sync}: & \forall((\alpha\ \mathbf{com}\ \beta_1) \rightarrow^{\beta_2} \alpha, [\beta_2 \sqsupseteq \beta_1\,]) \\
\mathsf{send}: & \forall(\alpha\ \mathbf{chan} \times \alpha \rightarrow^{\beta_1} \alpha\ \mathbf{com}\ \beta_2, [\beta_1 \sqsupseteq \epsilon, \beta_2 \sqsupseteq\, !\alpha\,]) \\
\mathsf{receive}: & \forall(\alpha\ \mathbf{chan} \rightarrow^{\beta_1} \alpha\ \mathbf{com}\ \beta_2, [\beta_1 \sqsupseteq \epsilon, \beta_2 \sqsupseteq\, ?\alpha\,]) \\
\mathsf{channel}: & \forall(\mathsf{unit} \rightarrow^{\beta} \alpha\ \mathbf{chan}, [\beta \sqsupseteq \alpha\,\mathsf{CHAN}\,]) \\
\mathsf{fork}: & \forall((\mathsf{unit} \rightarrow^{\beta_1} \alpha) \rightarrow^{\beta_2} \mathsf{unit}, \beta_2 \sqsupseteq \mathsf{FORK}\,\beta_1)
\end{array}
$$

The inference system is much as in (Nielson and Nielson, 1994a) (whereas the inference system in (Nielson and Nielson, 1993) uses subtyping instead of polymorphism).

4 Designing the reconstruction algorithm W

Our goal is to produce an algorithm which works in the spirit of the well-known algorithm W (Milner, 1978), but due to the additional features present in our inference system some complications arise as will be described in the subsequent sections. In Section 4.1 we introduce the notion of simple types which is needed since behaviours constitute a non-free algebra; in Section 4.2 we describe the approach of (Nielson and Nielson, 1994b) introducing the notion of S-constraints; in Section 4.3 we improve on this approach so as to get completeness; and in Section 4.4 we further improve on our algorithm by eliminating some redundancy in the generated constraints thus making the output simpler, at the same time providing the motivation for an alternative way to write type schemes to be presented in Section 4.5. After all these preparations, our algorithm W is presented in Section 4.6.

4.1 The need for simple types

Due to the laws in Figure 2 the behaviours do not constitute a free algebra and hence the standard compositional unification algorithm is not immediately applicable. To see this, notice that even though $b_1; b_2 \equiv b_1'; b_2'$ it does not necessarily hold that $b_1 \equiv b_1'$ since we might have that $b_1 = b_2' = \epsilon$ and $b_1' = b_2 = !\mathsf{int}$.

The remedy (Talpin and Jouvelot, 1992; Nielson and Nielson, 1994b) is to introduce the notion of *simplicity*: a type is simple if all the behaviours it contains are behaviour variables (so e.g. $t_1 \rightarrow^{b} t_2$ is simple if and only if 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 if and only if 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 $\beta \sqsupseteq b$ with b a simple behaviour; a substitution is simple if it maps type variables into simple types and maps behaviour variables into behaviour variables (rather than simple behaviours).

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{UNIFY}(\alpha, t) = \mathsf{UNIFY}(t, \alpha) = [\alpha \mapsto t]} \\
\quad \text{iff} & \alpha \notin \mathsf{fv}(t) \text{ or } t = \alpha \\
\multicolumn{2}{l}{\mathsf{UNIFY}(\mathsf{int}, \mathsf{int}) = \mathsf{UNIFY}(\mathsf{bool}, \mathsf{bool}) = \mathsf{UNIFY}(\mathsf{unit}, \mathsf{unit}) = id} \\
\multicolumn{2}{l}{\mathsf{UNIFY}(t_1\ \mathbf{list}, t_2\ \mathbf{list}) = \mathsf{UNIFY}(t_1\ \mathbf{chan}, t_2\ \mathbf{chan}) = \theta} \\
\quad \text{iff} & \mathsf{UNIFY}(t_1, t_2) = \theta \\
\multicolumn{2}{l}{\mathsf{UNIFY}(t_1\ \mathbf{com}\ \beta_1, t_2\ \mathbf{com}\ \beta_2) = \theta'; [\beta_1' \mapsto \beta_2']} \\
\quad \text{iff} & \mathsf{UNIFY}(t_1, t_2) = \theta' \text{ and } \beta_i' = \beta_i[\theta'] \\
\multicolumn{2}{l}{\mathsf{UNIFY}(t_1 \times t_1', t_2 \times t_2') = \theta'; \theta''} \\
\quad \text{iff} & \mathsf{UNIFY}(t_1, t_2) = \theta' \\
\quad \text{and} & \mathsf{UNIFY}(t_1'[\theta'], t_2'[\theta']) = \theta'' \\
\multicolumn{2}{l}{\mathsf{UNIFY}(t_1 \rightarrow^{\beta_1} t_1', t_2 \rightarrow^{\beta_2} t_2') = \theta'; \theta''; [\beta_1' \mapsto \beta_2']} \\
\quad \text{iff} & \mathsf{UNIFY}(t_1, t_2) = \theta' \\
\quad \text{and} & \mathsf{UNIFY}(t_1'[\theta'], t_2'[\theta']) = \theta'' \\
\quad \text{and} & \beta_i' = \beta_i[\theta'; \theta''] \\
\multicolumn{2}{l}{\mathsf{UNIFY}(t_1, t_2) \text{ fails otherwise}}
\end{array}
$$

Fig. 3. Procedure UNIFY.

Fact 4.1

Simple types are closed under the application of simple substitutions: $t[\phi]$ is simple if t and ϕ are; similarly for behaviours and C-constraints. Also simple substitutions are closed under composition: $\phi; \phi'$ (first ϕ and then ϕ') is simple if both ϕ and ϕ' are. □

Fact 4.2

All CTypeOf(c) have simple types and simple C-constraints. □

In Fig. 3 we define a procedure UNIFY which takes two simple types t_1 and t_2 and returns the most general unifier if a unifier exists – otherwise UNIFY fails. There are two different non-failing cases: (i) if one of the types is a variable, we return a unifying substitution after having performed an "occur check"; (ii) if both types are composite types with the same topmost constructor, we call UNIFY recursively on the type components and subsequently identify the behaviour components (which is possible since these have to be variables as the types are simple).

Fact 4.3

If UNIFY is called on simple types, all arguments to subcalls will be simple types and the substitution returned by UNIFY is simple. □

The following two lemmas state that UNIFY really computes the most general unifier:

Lemma 4.4

Suppose $\mathsf{UNIFY}(t_1, t_2) = \theta$ with t_1 and t_2 simple. Then $t_1[\theta] = t_2[\theta]$. □

Proof

Induction in the definition of UNIFY, using the same terminology. For the call $\mathsf{UNIFY}(\alpha, t)$ we have $\alpha[\alpha \mapsto t] = t = t[\alpha \mapsto t]$ where we employ that $\alpha \notin \mathsf{fv}(t)$ (or $t = \alpha$).

Next suppose $\mathsf{UNIFY}(t_1 \ \mathbf{com} \ \beta_1, t_2 \ \mathbf{com} \ \beta_2) = \theta$ with $\theta = \theta'; [\beta_1' \mapsto \beta_2']$. By induction we have $t_1[\theta'] = t_2[\theta']$, which implies $t_1[\theta] = t_2[\theta]$ and hence

$$(t_1 \ \mathbf{com} \ \beta_1)[\theta] = t_1[\theta] \ \mathbf{com} \ \beta_1'[\beta_1' \mapsto \beta_2'] = t_2[\theta] \ \mathbf{com} \ \beta_2' = (t_2 \ \mathbf{com} \ \beta_2)[\theta].$$

The remaining cases are similar. □

Lemma 4.5

Suppose $t_1[\psi] = t_2[\psi]$ with t_1 and t_2 simple. Then $\mathsf{UNIFY}(t_1, t_2)$ succeeds with result θ, and there exists ψ' such that $\psi = \theta; \psi'$. □

Proof

Induction in the sum of the sizes of t_1 and t_2. If one of these is a variable, then the claim follows from the fact that if $\alpha[\psi] = t[\psi]$ then $\alpha \notin \mathsf{fv}(t)$ or $\alpha = t$, and also $\psi = [\alpha \mapsto t]; \psi$.

Otherwise, they must have the same topmost constructor say **com** (the other cases are rather similar). That is, the situation is that $(t_1 \ \mathbf{com} \ \beta_1)[\psi] = (t_2 \ \mathbf{com} \ \beta_2)[\psi]$. Since $t_1[\psi] = t_2[\psi]$ we can apply the induction hypothesis to infer that the call $\mathsf{UNIFY}(t_1, t_2)$ succeeds with result θ' and that there exists ψ' such that $\psi = \theta'; \psi'$. With $\beta_1' = \beta_1[\theta']$, with $\beta_2' = \beta_2[\theta']$ and with $\theta = \theta'; [\beta_1' \mapsto \beta_2']$ we conclude that $\mathsf{UNIFY}(t_1 \ \mathbf{com} \ \beta_1, t_2 \ \mathbf{com} \ \beta_2)$ succeeds with result θ. Since $\beta_1'[\psi'] = \beta_1[\theta'; \psi'] = \beta_1[\psi] = \beta_2[\psi] = \beta_2[\theta'; \psi'] = \beta_2'[\psi']$ it holds that $\psi' = [\beta_1' \mapsto \beta_2']; \psi'$. Hence we have the desired relation $\psi = \theta'; \psi' = \theta; \psi'$. □

4.2 A previous approach with S-constraints

A (sound but not complete) reconstruction algorithm for the inference system in Fig. 1 was presented in (Nielson and Nielson, 1994b). Inspired by e.g. (Talpin and Jouvelot, 1992) the algorithm collected a set of (what we call) C-constraints and accordingly the environment mapped identifiers to *extended* type schemes (i.e. containing C-constraints), but in addition also a set of "S-constraints" had to be collected as will be explained below.

Consider the expression $\mathsf{let} \ x{=}e_1 \ \mathbf{in} \ e_0$ where x occurs twice in e_0; here e_0 must be analysed in an environment where x is bound to an extended type scheme $\forall\vec{\gamma}.(t, C)$

with C the constraints generated when analysing e_1. The first occurrence of x in e_0 gives rise to a copy of C (and t), where the polymorphic variables $\vec{\gamma}$ are replaced by fresh variables $\vec{\gamma}'$. Let C_0 be the constraints generated when analysing e_0; it is easy to see that given ψ_0 satisfying C_0 we can find a substitution ψ' which satisfies C by stipulating that $\vec{\gamma}[\psi'] = \vec{\gamma}'[\psi_0]$ and that ψ' equals ψ_0 on $\mathsf{fv}(\forall\vec{\gamma}.(t, C))$. With $\vec{\gamma}''$ the fresh variables generated by the second occurrence of x in e_0, we can find yet another substitution ψ'' satisfying C (now we stipulate $\vec{\gamma}[\psi''] = \vec{\gamma}''[\psi_0]$).

The problem is that ψ' and ψ'' are not necessarily related, even though they both satisfy C, as there seems to be no notion of "principal solutions" to C-constraints (this is unlike the situation in (Talpin and Jouvelot, 1994) where behaviours are sets of "atomic" effects). To see this, consider the constraint $(\beta \sqsupseteq (!\mathsf{int}; !\mathsf{int}; \beta)$. Both the substitution ψ_1 which maps β into $\mathsf{REC}\beta.(!\mathsf{int}; !\mathsf{int}; \beta)$ and the substitution ψ_2 which maps β into $\mathsf{REC}\beta.(!\mathsf{int}; \beta)$ will satisfy this constraint; but with the current axiomatisation it seems hard to find a sense in which ψ_1 and ψ_2 are comparable.[†]

So even though (as we shall see in Section 8) it is always possible to find *a* solution to a given set of C-constraints, such a solution may not correspond to a valid inference: in the example above concerning the typing of an expression $\mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0$, it may happen that the constraints on $\vec{\gamma}'$ and the constraints on $\vec{\gamma}''$ are solved in an "incompatible" way and hence the types assigned to the two occurrences of x in e_0 will not be instances of a common type scheme (to be assigned to e_1). In (Nielson and Nielson, 1994b) it is therefore required that a substitution ψ satisfying C_0 must also satisfy that $\vec{\gamma}'[\psi]$ and $\vec{\gamma}''[\psi]$ are instances of $\vec{\gamma}[\psi]$; this requirement is encoded in the form of an "S-constraint".

An additional feature present in (Nielson and Nielson, 1994b), needed in order for the soundness proof to carry through (and enforced by another kind of S-constraints), is that there is a sharp distinction between polymorphic variables and non-polymorphic variables in the sense that a solution should not "mix" these variables; in other words a solution ψ must satisfy that for every polymorphic variable γ and every non-polymorphic variable γ', the sets $\mathsf{fv}(\gamma[\psi])$ and $\mathsf{fv}(\gamma'[\psi])$ are disjoint. This requirement has severe impact on which variables to quantify (i.e. include in $\vec{\gamma}$) in the type scheme $\forall\vec{\gamma}.(t, C)$ of a let-bound identifier: apart from following the inference system in ensuring that variables free in the environment or in the behaviour are not quantified over, the approach of (Nielson and Nielson, 1994b) also needs to ensure that the set of variables not quantified over "respects C" — an equivalent formulation of this is that it must be *downwards closed* as well as *upwards closed* wrt. C, according to the following definitions:

[†] A remedy *might* be to adopt more rules for behaviours such that $\mathsf{REC}\beta.b$ is equivalent to its infinite unfolding (cf. rule R1 in Figure 2 which states that $\mathsf{REC}\beta.b$ is equivalent to its finite unfoldings, and cf. (Cardone and Coppo, 1991) where a similar change in axiomatisation is made concerning recursive types).

Definition 4.6

Let F be a set of variables and let C be a set of constraints. We say that F is *downwards closed* wrt. C if the following property holds for all $\beta \sqsupseteq b \in C$: if $\beta \in F$ then $\mathsf{fv}(b) \subseteq F$. □

Definition 4.7

Let F be a set of variables and let C be a set of constraints. We say that F is *upwards closed* wrt. C if the following property holds for all $\beta \sqsupseteq b \in C$: if $\mathsf{fv}(b) \cap F \neq \emptyset$ then $\beta \in F$. □

We define the downwards closure of F wrt. C, denoted $F^{\downarrow C}$, as the least set which contains F and which is downwards closed wrt. C. It is easy to see that this set can be computed constructively.

In the rest of the paper $\mathcal{NQ}$ denotes the set of variables not quantified over. Demanding $\mathcal{NQ}$ to be downwards closed amounts to stating that a non-polymorphic variable cannot have polymorphic subparts (which seems reasonable); whereas additionally demanding $\mathcal{NQ}$ to be upwards closed amounts to stating that a polymorphic variable cannot have non-polymorphic subparts (which seems overly demanding).

4.3 Achieving completeness

The last remarks in the preceding section suggest that the proper demand on $\mathcal{NQ}$ is that it must be downwards closed but not necessarily upwards closed. This modification is actually the key to getting an algorithm which is complete. But without $\mathcal{NQ}$ being upwards closed we cannot expect the existence of a solution which does not mix up polymorphic and non-polymorphic variables (cf. the previous discussion). Hence this restriction has to be weakened (but not completely abandoned), and this can be accomplished by letting S-constraints take the form $(\forall \overline{F}.\vec{g} \succ \vec{g}')$; here F is a set of variables which one should think of as *non-polymorphic*, and g ranges over types and behaviours collectively.

Definition 4.8

An S-constraint $\forall \overline{F}.\vec{g} \succ \vec{g}'$ is *satisfied* by ψ if and only if there exists an "instance substitution" ϕ, with $\mathsf{dom}(\phi)$ disjoint from $\mathsf{fv}(F[\psi])$, such that $\vec{g}'[\psi] = \vec{g}[\psi][\phi]$. □

This explains S-constraints as a special case of semi-unification (Henglein, 1993).

4.4 Eliminating redundancy

When meeting an identifier x which is bound to a type scheme $\forall \vec{\gamma}.(t, C)$ the algorithm should proceed as follows: the S-constraint $\forall \overline{F}.\vec{\gamma} \succ \vec{\gamma}'$ is generated where $\vec{\gamma}'$ are fresh copies of $\vec{\gamma}$ and where $F = \mathsf{fv}(t, C) \setminus \vec{\gamma}$; in addition copies of the C-constraints in C are generated (replacing $\vec{\gamma}$ by $\vec{\gamma}'$). There is some redundancy in

this and actually it is possible to dispense with copying the C-constraints. This in turn enables us to remove constraints from the type schemes. The virtues of doing so are twofold: the output from the implementation becomes much smaller; and the correctness proofs become simpler. The price to pay is that even though C can be removed from $\forall\vec{\gamma}.(t, C)$ we still have to remember what $\mathsf{fv}(C)$ is; otherwise F as defined above will become too small and hence the generated S-constraints will become too easy to satisfy, making the algorithm unsound.

4.5 Type schemes redefined

The considerations in the previous section suggest that it is convenient to write type schemes ts on the form $\forall\overline{F}.t$ where F is a list of *free* variables (the notation indicates that the set of bound variables is the "complement" of F). We thus define $\mathsf{fv}(ts) = F$; and say that ts is simple if t is.

There is a natural injection from type schemes in the classical form $\forall\vec{\gamma}.t$ into type schemes in the new form (let $F = \mathsf{fv}(t) \setminus \vec{\gamma}$). A type scheme $\forall\overline{F}.t$ which is in the image of this injection (i.e. where $F \subseteq \mathsf{fv}(t)$) is said to be *kernel*; type schemes which are not necessarily kernel are said to be *enriched*.

The instance relation is defined in a way consistent with the classical definition: $\forall\overline{F}.t \succ t'$ holds if and only if there exists an "instance substitution" ϕ with $\mathsf{dom}(\phi) \cap F = \emptyset$ such that $t' = t[\phi]$.

Concerning the function $\mathsf{gen}(t_1, E, b_1)$ employed in Fig. 1, the defining equation (1) is now written

$$\mathsf{gen}(t_1, E, b_1) = \forall\overline{F}.t_1 \text{ where } F = \mathsf{fv}(t_1) \cap (\mathsf{fv}(E) \cup \mathsf{fv}(b_1)). \tag{2}$$

We still need extended type schemes of form $\forall\overline{F}.(t, C)$, with C being C-constraints, to appear in CTypeOf(c); but here it will always be the case that $F = \emptyset$ and hence we simply write $\forall(t, C)$. As usual $\forall(t, C) \succ t'$ will hold if and only if there exists ϕ such that ϕ satisfies C and $t' = t[\phi]$.

Thus the inference system in Fig. 1, which operates on type schemes in classical form, can equivalently be considered as an inference system operating on kernel type schemes. The reconstruction algorithm, however, may encounter non-kernel type schemes.

We also need a relation $ts \succeq ts'$ (to be read: ts is more general than ts') on type schemes (to be extended pointwise to environments). Usually this is defined to hold if all instances of ts' are also instances of ts, but it turns out that for our purposes a *stronger* version will be more suitable (as it is more "syntactic"): with $ts = \forall\overline{F}.t$ and $ts' = \forall\overline{F'}.t'$ we say that $ts \succeq ts'$ holds if and only if $t = t'$ and $F \subseteq F'$. As expected we have

Fact 4.9

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

Finally we need to define how substitutions work on type schemes and S-constraints[‡] (one should read $\forall\overline{F}.t[\psi]$ as $\forall\overline{F}.(t[\psi])$ and *not* as $(\forall\overline{F}.t)[\psi]$, similarly one should read $\forall\overline{F}.\vec{g}[\psi] \succ \vec{g'}[\psi]$ as $\forall\overline{F}.(\vec{g}[\psi]) \succ (\vec{g'}[\psi])$):

Definition 4.10

If $ts = \forall\overline{F}.t$ then $ts[\psi] = \forall\overline{F'}.t[\psi]$ where $F' = \mathsf{fv}(F[\psi])$.
The result of applying ψ to the S-constraint $\forall\overline{F}.\vec{g} \succ \vec{g'}$ is $\forall\overline{F'}.\vec{g}[\psi] \succ \vec{g'}[\psi]$ where again $F' = \mathsf{fv}(F[\psi])$. □

Notice that the S-constraint $\forall\overline{F}.t \succ t'$ is satisfied by ψ (cf. Def. 4.8) if and only if the type $t'[\psi]$ is an instance of the (enriched) type scheme $(\forall\overline{F}.t)[\psi]$. In the following we shall often write $\psi \models C$ to denote that ψ satisfies C; and we shall often (silently) make use of the following observations:

Fact 4.11

Let ts be a type scheme, let C be a set of constraints, and let ψ and ψ' be substitutions. Then

- $\mathsf{fv}(\mathsf{fv}(ts)[\psi]) = \mathsf{fv}(ts[\psi])$;
- $ts[\psi;\psi'] = ts[\psi][\psi']$;
- $\psi;\psi' \models C$ holds if and only if $\psi' \models C[\psi]$.

4.6 Algorithm W

We are now ready to define the reconstruction algorithm W, as is done in Fig. 4 and Fig. 5. The algorithm fails if and only if a call to UNIFY fails. It takes as input a CML-expression and an environment binding identifiers to simple enriched type schemes; it returns as output a simple type, a simple behaviour, a list of constraints ($\oplus$ concatenates such lists) where the C-constraints are simple, and a simple substitution (all this can easily be verified).

Most parts of the algorithm are either standard or have been explained earlier in this section. Note that in the clause for constants we generate a copy of the C-constraints rather than an S-constraint, unlike what we do in the clause for identifiers. This corresponds to the difference in use: in an expression $\mathsf{let}\ x{=}e_1\ \mathbf{in}\ \ldots x \ldots x \ldots$ the types of the two x's must be instances of what we find *later* (when solving the generated constraints) to be the type of e_1; whereas in an expression $\ldots c \ldots c \ldots$ the types of the two c's must be instances of a type that we know *already*.

[‡] One should point out that in general it no longer holds that if $ts \succ t$ then $ts[\psi] \succ t[\psi]$ (consider e.g. $ts = \forall\overline{\alpha_2}.\alpha_1 \times \alpha_2$, $t = \mathsf{int} \times \alpha_2$ and $\psi = [\alpha_2 \mapsto \alpha_1 \times \alpha_2]$, where the "bound" variable α_1 occurs in the range of ψ). This may seem strange, but such a general result is not needed for our correctness proofs (only a restricted version, cf. Fact B.3); and the result does hold for substitutions as long as they do not affect the bound variables (we have sketched a proof that this will be the case for the substitutions produced by our algorithm).

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

$W(c, E) = (t, b, C, \theta)$
iff $\mathrm{CTypeOf}(c) = \forall(t_c, C_c)$
and $\vec{\gamma} = \mathsf{fv}(t_c) \cup \mathsf{fv}(C_c)$ and $\vec{\gamma}'$ are fresh copies of $\vec{\gamma}$
and $t = t_c[\vec{\gamma} \mapsto \vec{\gamma}']$ and $b = \epsilon$ and $C = C_c[\vec{\gamma} \mapsto \vec{\gamma}']$ and $\theta = id$

$W(e_1\ e_2, E) = (t, b, C, \theta)$
iff $W(e_1, E) = (t_1, b_1, C_1, \theta_1)$ and $W(e_2, E[\theta_1]) = (t_2, b_2, C_2, \theta_2)$
and α and β_0 are fresh and $\mathsf{UNIFY}(t_1[\theta_2], t_2 \rightarrow^{\beta_0} \alpha) = \theta_0$
and $t = \alpha[\theta_0]$ and $b = b_1[\theta_2; \theta_0]; b_2[\theta_0]; \beta_0[\theta_0]$
and $C = C_1[\theta_2; \theta_0] \oplus C_2[\theta_0]$ and $\theta = \theta_1; \theta_2; \theta_0$

$W(\lambda x.e_0, E) = (t, b, C, \theta)$
iff α_1 is a fresh variable and $W(e_0, E \oplus [x : \alpha_1]) = (t_0, b_0, C_0, \theta_0)$
and β_0 is a fresh variable and $t = \alpha_1[\theta_0] \rightarrow^{\beta_0} t_0$ and $b = \epsilon$
and $C = C_0 \oplus [\beta_0 \sqsupseteq b_0]$ and $\theta = \theta_0$

Fig. 4. Algorithm W, first part.

Properties of $\mathcal{NQ}$: In Fig. 5 we defined

$$\mathcal{NQ} = \mathcal{EB}^{\downarrow C_1}$$

(with $\mathcal{EB} = \mathsf{fv}(E[\theta_1]) \cup \mathsf{fv}(b_1)$) and it is easy to see that then $\mathcal{NQ}$ will satisfy (3) and (4) below (the latter is a consequence of Fact 3.2):

$$\psi \models C_1 \quad \Rightarrow \quad \mathsf{fv}(\mathcal{NQ}[\psi]) \supseteq \mathsf{fv}(\mathcal{EB}[\psi]) \tag{3}$$

$$\psi \models C_1 \quad \Rightarrow \quad \mathsf{fv}(\mathcal{NQ}[\psi]) \subseteq \mathsf{fv}(\mathcal{EB}[\psi]) \tag{4}$$

It turns out that in order to prove soundness of W, all we need to know about $\mathcal{NQ}$ is that it satisfies (3); and it turns out that in order to prove completeness of W, all we need to know about $\mathcal{NQ}$ is that it satisfies (4).

5 Soundness of algorithm W

We shall prove that the algorithm is sound; i.e. that a solution to the constraints gives rise to a valid inference in the inference system of Fig. 1.

Theorem 5.1

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

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

$W(\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, E) = (t, b, C, \theta)$
iff $W(e_0, E) = (t_0, b_0, C_0, \theta_0)$
and $W(e_1, E[\theta_0]) = (t_1, b_1, C_1, \theta_1)$
and $W(e_2, E[\theta_0;\theta_1]) = (t_2, b_2, C_2, \theta_2)$
and $\mathsf{UNIFY}((t_0[\theta_1;\theta_2] \times t_1[\theta_2]), (\mathsf{bool} \times t_2)) = \theta'$
and $t = t_2[\theta']$ and $b = (b_0[\theta_1;\theta_2];\ (b_1[\theta_2] + b_2))[\theta']$
and $C = (C_0[\theta_1;\theta_2]{\oplus}C_1[\theta_2]{\oplus}C_2)[\theta']$ and $\theta = \theta_0;\theta_1;\theta_2;\theta'$

$W(\mathsf{rec}\ f(x) \Rightarrow e_0, E) = (t, b, C, \theta)$
iff α_1, α_2 and β are fresh variables
and $W(e_0, E \oplus [f : \alpha_1 \rightarrow^{\beta} \alpha_2] \oplus [x : \alpha_1]) = (t_0, b_0, C_0, \theta_0)$
and $\mathsf{UNIFY}(\alpha_2[\theta_0], t_0) = \theta'$
and $t = (\alpha_1[\theta_0] \rightarrow^{\beta[\theta_0]} t_0)[\theta']$ and $b = \epsilon$
and $C = (C_0{\oplus}[\beta[\theta_0]{\sqsupseteq}b_0\,])[\theta']$ and $\theta = \theta_0;\theta'$

$W(e_1;e_2, E) = (t, b, C, \theta)$
iff $W(e_1, E) = (t_1, b_1, C_1, \theta_1)$ and $W(e_2, E[\theta_1]) = (t_2, b_2, C_2, \theta_2)$
and $t = t_2$ and $b = b_1[\theta_2];\ b_2$ and $C = C_1[\theta_2]{\oplus}C_2$ and $\theta = \theta_1;\theta_2$

Fig. 5. Algorithm W, second part.

This theorem follows easily (using Fact 3.1) from Proposition 5.2 below that admits an inductive proof. The formulation makes use of a function $\kappa(E)$ which maps enriched environments (as used by the algorithm) into kernel environments (as used in the inference system): for a type scheme $ts = \forall\overline{F}.t$ we define $\kappa(ts) = \forall\overline{F'}.t$ where $F' = F \cap \mathsf{fv}(t)$.

Proposition 5.2

Suppose that $W(e, E) = (t, b, C, \theta)$ with E simple.
Then for all ψ with $\psi \models C$ we have $\kappa(E[\theta][\psi]) \vdash e : t[\psi]\ \&\ b[\psi]$. □

Proof

The proof is by induction on e; to conserve space we use the same terminology as in the definition of the relevant clause for W. We perform a case analysis on the form of e; the cases for $\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$, $\mathsf{rec}\ f(x) \Rightarrow e_0$, and $e_1;e_2$ are omitted as they present no further complications.

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

We must show that $\kappa(E[\psi])(x) \succ t[\psi]$ which amounts to

$$\forall \overline{F'_x}.(t_x[\psi]) \succ t_x[\vec{\gamma} \mapsto \vec{\gamma}'][\psi]. \tag{5}$$

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 $\psi;\phi'$ equals $[\vec{\gamma} \mapsto \vec{\gamma}'];\psi$ on $\vec{\gamma}$. This implies, since $\mathsf{fv}(t_x) \subseteq \vec{\gamma} \cup F_x$, that we even have that $\psi;\phi'$ equals $[\vec{\gamma} \mapsto \vec{\gamma}'];\psi$ on $\mathsf{fv}(t_x)$. But this shows that (5) holds, with ϕ' as the "instance substitution".

The case $W(c,E)$: Since $\psi \models C$ we have $[\vec{\gamma} \mapsto \vec{\gamma}'];\psi \models C_c$ so it holds that $\forall(t_c, C_c) \succ t_c[[\vec{\gamma} \mapsto \vec{\gamma}'];\psi]$. Therefore we have the inference

$$\kappa(E[\theta][\psi]) \ \vdash \ c : t_c[[\vec{\gamma} \mapsto \vec{\gamma}'];\psi] \ \& \ \epsilon$$

which amounts to the desired relation.

The case $W(e_1\ e_2,E)$: Since $\psi \models C$ it holds that $\theta_2;\theta_0;\psi \models C_1$ so we can apply the induction hypothesis on the call $W(e_1,E)$ and the substitution $\theta_2;\theta_0;\psi$ to get

$$\kappa(E[\theta][\psi]) \ \vdash \ e_1 : t_1[\theta_2;\theta_0;\psi] \ \& \ b_1[\theta_2;\theta_0;\psi].$$

which by the soundness of UNIFY (Lemma 4.4) amounts to

$$\kappa(E[\theta][\psi]) \ \vdash \ e_1 : t_2[\theta_0;\psi] \rightarrow^{\beta_0[\theta_0;\psi]} \alpha[\theta_0;\psi] \ \& \ b_1[\theta_2;\theta_0;\psi].$$

As it moreover holds that $\theta_0;\psi \models C_2$ we can apply the induction hypothesis on the call $W(e_2,E[\theta_1])$ and the substitution $\theta_0;\psi$ to get

$$\kappa(E[\theta][\psi]) \ \vdash \ e_2 : t_2[\theta_0;\psi] \ \& \ b_2[\theta_0;\psi].$$

The last two judgements enable us to arrive at the desired judgement

$$\kappa(E[\theta][\psi]) \ \vdash \ e_1\ e_2 : t[\psi] \ \& \ b[\psi].$$

The case $W(\lambda x.e_0,E)$: Since $\psi \models C_0$ we can apply the induction hypothesis to get

$$\kappa(E[\theta_0][\psi]) \oplus [x : \alpha_1[\theta_0][\psi]] \ \vdash \ e_0 : t_0[\psi] \ \& \ b_0[\psi].$$

By Fact 3.1 and the fact that $\beta_0[\psi] \sqsupseteq b_0[\psi]$, we therefore have

$$\kappa(E[\theta_0][\psi]) \oplus [x : \alpha_1[\theta_0][\psi]] \ \vdash \ e_0 : t_0[\psi] \ \& \ \beta_0[\psi].$$

This shows the desired judgement

$$\kappa(E[\theta_0][\psi]) \ \vdash \ \lambda x.e_0 : (\alpha_1[\theta_0] \rightarrow^{\beta_0} t_0)[\psi] \ \& \ \epsilon.$$

The case $W(\mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0,E)$: Since $\theta_0;\psi \models C_1$ we can apply the induction hypothesis on the call $W(e_1,E)$ and the substitution $\theta_0;\psi$ to get

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

In order to arrive at the desired judgement

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

we must show that

$$\kappa(E[\theta][\psi]) \oplus [x : \forall\overline{F''}.t_1[\theta_0;\psi]] \ \vdash \ e_0 : t_0[\psi] \ \& \ b_0[\psi] \tag{6}$$

where $F'' = (\mathsf{fv}(\kappa(E[\theta][\psi])) \cup \mathsf{fv}(b_1[\theta_0;\psi])) \cap \mathsf{fv}(t_1[\theta_0;\psi])$.

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

$$\kappa(E[\theta][\psi]) \oplus [x : \forall\overline{F'}.(t_1[\theta_0;\psi])] \ \vdash \ e_0 : t_0[\psi] \ \& \ b_0[\psi] \tag{7}$$

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

We can infer (6) from (7) by Fact 4.9, provided that $F'' \subseteq F'$. But this follows from the calculation below, where we for the last equality use (3) (cf. Section 4.6) on the substitution $\theta_0;\psi$ (which satisfies C_1):

$$\begin{aligned}
&\mathsf{fv}(\kappa(E[\theta][\psi])) \cup \mathsf{fv}(b_1[\theta_0;\psi]) \\
&\quad\subseteq \quad \mathsf{fv}(E[\theta][\psi]) \cup \mathsf{fv}(b_1[\theta_0;\psi]) = \mathsf{fv}((\mathsf{fv}(E[\theta_1]) \cup \mathsf{fv}(b_1))[\theta_0;\psi]) = \mathsf{fv}(\mathcal{EB}[\theta_0;\psi]) \\
&\quad\subseteq \quad \mathsf{fv}(\mathcal{NQ}[\theta_0;\psi]).
\end{aligned}$$

This concludes the proof of Proposition 5.2. □

6 Completeness of algorithm W

We shall prove that if there exists a valid inference then the algorithm will produce a set of constraints which can be satisfied. This can be formulated in a way which is symmetric to Theorem 5.1:

Theorem 6.1

Suppose $\emptyset \ \vdash \ e : t' \ \& \ b'$.
Then $W(e, \emptyset)$ succeeds with result (t, b, C, θ)
and there exists ψ such that $\psi \models C$ and $t' = t[\psi]$ and $b' \sqsupseteq b[\psi]$. □

We have not succeeded in finding a direct proof of this result so our path will be (i) to define an inference system which is equivalent to the one in Fig. 1, and (ii) to prove the algorithm complete wrt. this inference system.

The problem with the original system is that generalisation is defined as in (2) with $\mathsf{gen}(t_1, E, b_1) = \forall\overline{F}.t_1$ where $F = \mathsf{fv}(t_1) \cap (\mathsf{fv}(E) \cup \mathsf{fv}(b_1))$; this is in contrast to the algorithm where no intersection with $\mathsf{fv}(t_1)$ is taken. This motivates the design of an alternative inference system which is as the original one except that it employs an alternative generalisation function:

$$\mathsf{gen}_2(t_1, E, b_1) = \forall\overline{F}.t_1 \text{ where } F = \mathsf{fv}(E) \cup \mathsf{fv}(b_1). \tag{8}$$

Hence inferences in this system will be of form $E \ \vdash_2 \ e : t \ \& \ b$ where the environment E may now contain *enriched* type schemes. We now have the desired equivalence result, to be proved in Appendix B:

Proposition 6.2

Assume $\kappa(E') = E$. Then $E \vdash e : t \;\&\; b$ holds if and only if $E' \vdash_2 e : t \;\&\; b$.
(In particular, $\emptyset \vdash e : t \;\&\; b$ holds if and only if $\emptyset \vdash_2 e : t \;\&\; b$.) □

So in order to prove Theorem 6.1 it will be sufficient to show Proposition 6.3 below that admits an inductive proof. Often (as in e.g. (Jones, 1992)) the assumptions in a completeness proposition are (using the terminology of Prop. 6.3) that $E[\phi]$ is *equal to* E'; but as in (Smith, 1993) this is not sufficient here since in the inference system an identifier may be bound to a type scheme which is less general than the one to which it is bound in the algorithm. In our system this phenomenon is due to the presence of subeffecting in the inference system, allowing variables which could otherwise be generalised to appear "superfluously" in the behaviour thus preventing generalisation; this is unlike (Smith, 1993) where it is due to the fact that the inference system gives freedom to quantify over fewer variables than possible.

In the formulation of Proposition 6.3 we write $\phi_1 \stackrel{E}{=} \phi_2$ to denote that $\gamma[\phi_1] = \gamma[\phi_2]$ for all $\gamma \in \mathsf{var}(E)$, where $\mathsf{var}(E)$ contains *all* the variables occurring in E: that is, $\mathsf{var}(E)$ is the union over $\mathsf{dom}(E)$ of $\mathsf{var}(E(x))$ where $\mathsf{var}(\forall \overline{F}.t) = \mathsf{fv}(t) \cup F$. For a substitution ϕ we define $\mathsf{var}(\phi) = \mathsf{fv}(\phi) = \mathsf{dom}(\phi) \cup \mathsf{ran}(\phi)$.

Proposition 6.3

Suppose $E' \vdash_2 e : t' \;\&\; b'$ and $E[\phi] \succeq E'$ with E simple.
Then $W(e, E)$ succeeds with result (t, b, C, θ) (all simple)
and there exists a ψ such that $\theta; \psi \stackrel{E}{=} \phi$ and $\psi \models C$ and $t' = t[\psi]$ and $b' \sqsupseteq b[\psi]$. □

Proof

The proof is by induction on the proof 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.

We perform a case analysis on the form of e; the cases for $\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$, $\mathsf{rec}\ f(x) \Rightarrow e_0$, and $e_1;e_2$ are omitted as they present no further complications.

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 $\mathsf{fv}(F_x[\phi]) \subseteq F'_x$. From this we infer that

$$\mathsf{dom}(\phi') \cap \mathsf{fv}(F_x[\phi]) = \emptyset. \tag{9}$$

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 from (9) we get

$$[\vec{\gamma} \mapsto \vec{\gamma}']; \psi \text{ equals } \phi; \phi' \text{ on } F_x \cup \vec{\gamma}. \tag{10}$$

Since ψ equals ϕ on $F_x \cup \vec{\gamma}$ this implies $\forall \overline{F_x[\psi]}.\vec{\gamma}[\psi] \;\succ\; \vec{\gamma}'[\psi]$ (with ϕ' as the instance

substitution) which amounts to $\psi \models C$. Finally, since $\mathsf{fv}(t_x) \subseteq F_x \cup \vec{\gamma}$ we also get from (10) that $t' = t'_x[\phi'] = t_x[\phi;\phi'] = t_x[[\vec{\gamma} \mapsto \vec{\gamma}'];\psi] = t[\psi]$.

The case $e = c$: Let $\mathrm{CTypeOf}(c) = \forall(t_c, C_c)$. Suppose $E' \vdash_2 c : t' \;\&\; b'$ holds because $b' \sqsupseteq \epsilon$ and because there exists a ψ' with $\psi' \models C_c$ such that $t' = t_c[\psi']$. Now define ψ as follows: it maps $\vec{\gamma}'$ into $\vec{\gamma}[\psi']$; and otherwise it behaves like ϕ. We have

$$t[\psi] = t_c[\psi'] \text{ and } C[\psi] = C_c[\psi']$$

which shows that $t' = t[\psi]$ and that $\psi \models C$. It is trivial that $\theta;\psi \stackrel{E}{=} \phi$ and that $b' \sqsupseteq b[\psi]$.

The case $e = e_1\ e_2$: Suppose $E' \vdash_2 e_1\ e_2 : t' \;\&\; b'$ holds because $E' \vdash_2 e_1 : t'_1 \;\&\; b'_1$, because $E' \vdash_2 e_2 : t'_2 \;\&\; b'_2$, and because there exists b'_0 such that $t'_1 = t'_2 \rightarrow^{b'_0} t'$ and $b' \sqsupseteq b'_1; b'_2; b'_0$.

By induction we see that the call $W(e_1, E)$ 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 $t'_1 = t_1[\psi_1]$, and such that $b'_1 \sqsupseteq b_1[\psi_1]$.

Since $E[\theta_1][\psi_1] = E[\phi]$ we infer that $E[\theta_1][\psi_1] \succeq E'$. Thus we can apply the induction hypothesis once more to infer that the call $W(e_2, E[\theta_1])$ succeeds and that there exists ψ_2 such that $\theta_2;\psi_2$ equals ψ_1 on $\mathsf{var}(E[\theta_1])$, such that $\psi_2 \models C_2$, such that $t'_2 = t_2[\psi_2]$, and such that $b'_2 \sqsupseteq b_2[\psi_2]$.

Some terminology: let $V_1 = \mathsf{var}(t_1, b_1, C_1, \theta_1)$ and $V_2 = \mathsf{var}(t_2, b_2, C_2, \theta_2)$ and $E_1 = \mathsf{var}(E[\theta_1])$; we shall say that a variable is "internal" if it occurs in V_1 but not in E_1. As the algorithm always picks fresh variables, no internal variable occurs in V_2, in particular not in $\mathsf{dom}(\theta_2)$ or in $\mathsf{ran}(\theta_2)$.

Now define ψ_0 to behave as ψ_2 except that it behaves as ψ_1 on internal variables; it maps α into t'; and it maps β_0 into b'_0.

We have the following relations:

$$\psi_0 \stackrel{V_2}{=} \psi_2 \text{ and } \theta_2;\psi_0 \stackrel{V_1 \cup E_1}{=} \psi_1 \tag{11}$$

where the second part follows from the following reasoning: if γ is internal then $\gamma[\theta_2;\psi_0] = \gamma[\psi_0] = \gamma[\psi_1]$; and if γ is not internal (and hence belongs to E_1) then $\gamma[\theta_2]$ does not contain any internal variables so $\gamma[\theta_2;\psi_0] = \gamma[\theta_2;\psi_2] = \gamma[\psi_1]$.

From (11) we infer that

$$t_1[\theta_2][\psi_0] = t_1[\psi_1] = t'_1 = t'_2 \rightarrow^{b'_0} t' = (t_2 \rightarrow^{\beta_0} \alpha)[\psi_0]$$

which by the completeness of UNIFY (Lemma 4.5) implies that the call to UNIFY succeeds (thus the call $W(e_1\ e_2, E)$ succeeds) and that there exists ψ such that $\psi_0 = \theta_0;\psi$. Using this and (11) we can infer the desired properties of ψ:

- If $\gamma \in \mathsf{var}(E)$ then $\gamma[\theta;\psi] = \gamma[\theta_1][\theta_2][\theta_0;\psi] = \gamma[\theta_1][\theta_2][\psi_0] = \gamma[\theta_1][\psi_1] = \gamma[\phi]$.
- To show that $\psi \models C$ holds we must show $\theta_2;\theta_0;\psi \models C_1$ and $\theta_0;\psi \models C_2$ which follows from $\psi_1 \models C_1$ and $\psi_2 \models C_2$.

- $t' = \alpha[\psi_0] = \alpha[\theta_0;\psi] = t[\psi]$.
- Since $\sqsupseteq$ is a pre-congruence (Rule C1 in Figure 2) we infer that

$$b' \sqsupseteq b'_1; b'_2; b'_0 \sqsupseteq b_1[\psi_1]; b_2[\psi_2]; \beta_0[\psi_0] = b_1[\theta_2][\psi_0]; b_2[\psi_0]; \beta_0[\psi_0] = b[\psi].$$

The case $e = \lambda x.e_0$*:* Suppose $E' \vdash_2 \lambda x.e_0 : t'$ & b' holds because we with $t' = t'_1 \rightarrow^{b'_0} t'_0$ and $b' \sqsupseteq \epsilon$ have $E' \oplus [x : t'_1] \vdash_2 e_0 : t'_0$ & b'_0. Define ϕ_0 to behave like ϕ except that it maps α_1 into t'_1. Then we clearly have

$$(E \oplus [x : \alpha_1])[\phi_0] \succeq E' \oplus [x : t'_1]$$

so by induction we see that the call $W(e_0, E \oplus [x : \alpha_1])$ succeeds and that there exists ψ_0 such that $\theta_0;\psi_0 \stackrel{E\cup\{\alpha_1\}}{=} \phi_0$; such that $\psi_0 \models C_0$; such that $t'_0 = t_0[\psi_0]$ and such that $b'_0 \sqsupseteq b_0[\psi_0]$.

Define ψ as follows: it maps β_0 into b'_0; and otherwise it behaves like ψ_0. It is obvious that $\theta;\psi \stackrel{E}{=} \phi$ and that $\psi \models C_0$. Since it moreover holds that

$$\beta_0[\psi] = b'_0 \sqsupseteq b_0[\psi]$$

we conclude that $\psi \models C$. Clearly $b' \sqsupseteq b[\psi]$, and finally we have

$$t' = t'_1 \rightarrow^{b'_0} t'_0 = \alpha_1[\phi_0] \rightarrow^{\beta_0[\psi]} t_0[\psi_0] = (\alpha_1[\theta_0] \rightarrow^{\beta_0} t_0)[\psi].$$

The case $e = \mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0$*:* 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 $t'_1 = t_1[\psi_1]$, and such that $b'_1 \sqsupseteq b_1[\psi_1]$.

In order to apply the induction hypothesis once more we must show

$$(E[\theta_1] \oplus [x : \forall\overline{\mathcal{NQ}}.t_1])[\psi_1] \succeq E' \oplus [x : \forall\overline{F'}.t'_1]. \tag{12}$$

But this is an easy consequence of the calculation

$$\begin{array}{rcl} \mathsf{fv}(\mathcal{NQ}[\psi_1]) & \subseteq & \mathsf{fv}((\mathsf{fv}(E[\theta_1]) \cup \mathsf{fv}(b_1))[\psi_1]) \\ & = & \mathsf{fv}(E[\phi]) \cup \mathsf{fv}(b_1[\psi_1]) \\ & \subseteq & \mathsf{fv}(E') \cup \mathsf{fv}(b_1[\psi_1]) \\ & \subseteq & \mathsf{fv}(E') \cup \mathsf{fv}(b'_1) = F' \end{array}$$

where the first inclusion follows from (4) (cf. Section 4.6) used on ψ_1; where the second inclusion follows from the assumption $E[\phi] \succeq E'$; and where the last inclusion follows from Fact 3.2 (this inclusion may be strict and therefore we need the predicate $\succeq$ rather than just equality, cf. the discussion prior to the proposition).

We have proved (12) so by induction we see that $W(e_0, \_)$ succeeds and that there exists ψ_0 such that $\theta_0;\psi_0$ equals ψ_1 on V_e where $V_e = \mathsf{var}(E[\theta_1]) \cup \mathcal{NQ} \cup \mathsf{fv}(t_1)$, such that $\psi_0 \models C_0$, such that $t' = t_0[\psi_0]$ and such that $b'_0 \sqsupseteq b_0[\psi_0]$.

Then some terminology (similar to the one introduced in the case for application):

let $V_1 = \mathsf{var}(t_1, b_1, C_1, \theta_1)$ and $V_0 = \mathsf{var}(t_0, b_0, C_0, \theta_0)$; we shall say that a variable is "internal" if it occurs in V_1 but not in V_e. As the algorithm always picks fresh variables, no internal variable occurs in V_0.

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

$$\psi \stackrel{V_0}{=} \psi_0 \text{ and } \theta_0;\psi \stackrel{V_1 \cup V_e}{=} \psi_1 \tag{13}$$

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 (13) 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) $t' = t_0[\psi_0] = t[\psi]$; (iv) we infer that $b' \sqsupseteq b'_1$; $b'_0 \sqsupseteq b_1[\psi_1]$; $b_0[\psi_0] = b_1[\theta_0][\psi]$; $b_0[\psi] = b[\psi]$. □

7 Choice of generalisation strategy

Recall that in Fig. 5 we defined $\mathcal{NQ}$ as $\mathcal{EB}^{\downarrow C_1}$ and we saw that to prove soundness it is sufficient to know that $\mathcal{NQ}$ satisfies (3):

$$\psi \models C_1 \Rightarrow \mathsf{fv}(\mathcal{NQ}[\psi]) \supseteq \mathsf{fv}(\mathcal{EB}[\psi])$$

and to prove completeness it is sufficient to know that $\mathcal{NQ}$ satisfies (4):

$$\psi \models C_1 \Rightarrow \mathsf{fv}(\mathcal{NQ}[\psi]) \subseteq \mathsf{fv}(\mathcal{EB}[\psi]).$$

In this section we shall investigate whether other definitions of $\mathcal{NQ}$ might be appropriate.

Requiring $\mathcal{NQ}$ to be upwards closed? (In addition to being downwards closed and contain $\mathcal{EB}$; as already mentioned this is essentially what is done in (Nielson and Nielson, 1994b).) Then (3) will still hold so soundness is assured. On the other hand (4) does not hold; in fact completeness fails since there exists well-typed CML-expressions on which the algorithm fails, e.g. the expression below:

```
λx. let f = λy. let ch1 = channel () in let ch2 = channel ()
                in λh.((sync (send ⟨ch1,x⟩));
                       (sync (send ⟨ch2,y⟩))))
                ; y
    in f 7;f true
```

which is typable since with $E = [x : \alpha_x]$ we have

$$E \vdash \lambda y.\ldots : \alpha_y \rightarrow^{\alpha_x} \mathsf{CHAN};\ \alpha_y\, \mathsf{CHAN}\ \alpha_y\ \&\ \epsilon$$

and hence it is possible to quantify over α_y. On the other hand, when analysing $\lambda y.\ldots$ the algorithm will generate constraints whose "transitive closure" includes

something like

$$\beta \sqsupseteq \alpha_x\,\mathsf{CHAN};\ \alpha_y\,\mathsf{CHAN}$$

and since α_x belongs to $\mathcal{EB}$ and hence to $\mathcal{NQ}$ also α_y will be in $\mathcal{NQ}$.

Not requiring $\mathcal{NQ}$ to be downwards closed? (So $\mathcal{NQ} = \mathcal{EB}$.) It is trivial that (3) and (4) still hold and hence neither soundness nor completeness is destroyed. 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. The proposed version of W applied to e will terminate successfully and return constraints including the following

$$[\beta \sqsupseteq \alpha\,\mathsf{CHAN},\ \forall\overline{\{\beta\}}.\alpha\ \succ\ \mathsf{bool},\ \forall\overline{\{\beta\}}.\alpha\ \succ\ \mathsf{int}\,]$$

which are unsolvable since for a solution substitution ψ it will hold (with $B = \mathsf{fv}(\beta[\psi])$) that $\forall\overline{B}.\alpha[\psi]\ \succ\ \mathsf{bool}$ and $\forall\overline{B}.\alpha[\psi]\ \succ\ \mathsf{int}$; in addition we have $\mathsf{fv}(\alpha[\psi]) \subseteq B$ so it even holds that $\alpha[\psi] = \mathsf{bool}$ and $\alpha[\psi] = \mathsf{int}$. On the other hand, the algorithm from Fig. 4 and 5 applied to e will fail immediately (since α is considered non-polymorphic and hence is not copied, causing UNIFY to fail). So it seems that the proposed change ought to be rejected on the basis that failures should be reported as early as possible.

The approach in (Talpin and Jouvelot, 1994).

We have seen that there are several possibilities for satisfying (3) and (4); our decision to use the downwards closure may seem somewhat arbitrary, but it can be justified by observing the similarities to (Talpin and Jouvelot, 1994). Here behaviours are *sets* of atomic "effects" (thus losing causality information) and any solvable constraint set C has a "canonical" solution $\overline{C}$ which is principal in the sense that for any ψ satisfying C it holds that $\psi = \overline{C};\psi$ (so defining $\mathcal{NQ}$ as $\mathsf{fv}(\mathcal{EB}[\overline{C_1}])$ will establish (3) and (4)). Essentially $\overline{C}$ maps β to $B \cup \{\beta\}$ if $(\beta \supseteq B)$ is in C; our $\mathcal{EB}^{\downarrow C_1}$ therefore corresponds to the $\mathsf{fv}(\mathcal{EB}[\overline{C_1}])$ already used in (Talpin and Jouvelot, 1994).

8 Solving the constraints

In this section we discuss how to solve the constraints generated by Algorithm W. We have seen that the C-constraints are simple, i.e. of form $\beta \sqsupseteq b$ with b a simple behaviour; we have sketched a proof that the S-constraints are of form $\forall\overline{F}.\vec{\alpha}\vec{\beta}\ \succ\ \vec{t}\,\vec{\beta}'$ where $\vec{\alpha}$ and $\vec{\beta}$ are vectors of 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 Figure 2 and it often decreases the size of behaviours significantly. The result of running our implementation on the program in Example 2.1 is depicted in Appendix A: only C-constraints are generated (as there are no

polymorphic variables), $>$ stands for $\sqsupseteq$, e stands for ϵ, the r_i are "region variables" and can be ignored.

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 ψ' satisfies C' and suppose ψ'' satisfies $C''[\psi']$. Then return $\psi = \psi'; \psi''$.

It is easy to see that $\psi \models C$ provided $C'[\psi']$ is such that $\phi \models C'[\psi']$ holds for all ϕ. This will be the case if $C'[\psi']$ only contains C-constraints (due to Fact 3.2) and S-constraints of form $\forall\overline{F}.\vec{g} \succ \vec{g}$ (where the two occurrences of $\vec{g}$ are equal), the latter kind to be denoted S-equalities. So we arrive at the following sufficient condition for when "sequential solving" is correct:

$$\text{only solve S-constraints when they turn into S-equalities.} \tag{14}$$

To see why sequential solving may go wrong if (14) is not imposed consider the constraints below:

$$\beta \sqsupseteq !\alpha,\ \beta' \sqsupseteq !\alpha,\ \forall\overline{\emptyset}.(\alpha,\beta) \succ (\mathsf{int},\beta'),\ \forall\overline{\emptyset}.(\alpha,\beta) \succ (\mathsf{bool},\beta'). \tag{15}$$

The two S-constraints are solved (but not into S-equalities!) by the identity substitution; so if we proceed sequentially we are left with the two C-constraints which are solved by the substitution ψ which maps β as well as β' into $!\alpha$. One might thus be tempted to think that ψ is a solution to the constraints in (15); but by applying ψ to these constraints we get

$$\forall\overline{\emptyset}.(\alpha,!\alpha) \succ (\mathsf{int},!\alpha),\ \forall\overline{\emptyset}.(\alpha,!\alpha) \succ (\mathsf{bool},!\alpha)$$

and these constraints are easily seen to be unsolvable.

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

First we identify all type and behaviour variables occurring in "corresponding positions" (i.e. unify all pairs $(\vec{\alpha}_i,\vec{\alpha}'_i)$ and $(\vec{\beta}_i,\vec{\beta}'_i)$), and in this way "the S-constraints are turned into S-equalities" (cf. (14)). Notice that by doing so, we abandon all polymorphism.

Next we have to solve the C-constraints sequentially; and during this process we want to preserve the invariant that they are of form $\beta \sqsupseteq b$ (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 $\{\beta \sqsupseteq b_1,\ldots,\beta \sqsupseteq b_n\}$ by means of a substitution whose domain is $\{\beta\}$. But this

can easily be achieved by adopting the canonical solution of (Nielson and Nielson, 1994b): due to rule R1 in Figure 2 we 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).

Our system implements the abovementioned (nondeterministically specified) algorithm; and when run on the program from Example 2.1 it produces:

```
*** Selected solution: ***
Type:      ((a4 -b17-> a14) -e-> (a4_list -b2-> a14_list))
Behaviour: e
where      b2 -> rec b2.(e+(r2_chan_a14_list;fork_((b2;r2!a14_list));
                           b17;r2?a14_list))
```

which (modulo renaming) is what we expect.

Solving constraints in the general case. One can encode S-constraints as a semi-unification problem and though the latter problem is undecidable several decidable subclasses exist; so one might be tempted to use e.g. the algorithm for semi-unification described in (Henglein, 1993). However, our problem is somewhat more complex because we also must solve the C-constraints, and as witnessed by the constraints in (15) this may destroy the solution to the S-constraints.

9 Conclusion

We have adapted the traditional algorithm W to our type and behaviour system. We have improved upon a previously published algorithm (Nielson and Nielson, 1994b) in achieving completeness and eliminating some redundancy in the representation of the constraints. 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 a phenomenon found also in (Jouvelot and Gifford, 1991).

Another and major difference from the traditional formulation, as well as that of (Jouvelot and Gifford, 1991), 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. This is not the case for the traditional "free" unification of Standard ML, but it is a phenomenon well-known in unification theory (Siekmann, 1989). 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. Solving S-constraints is a special case of semi-unification and even though the latter is undecidable we may hope the former to be decidable. At present it is an open problem how hard it is to solve S-constraints in the presence of C-constraints. This problem is closely

related to the question of whether the algorithm may generate constraints which cannot be solved.

Acknowledgements. This research has been supported by the DART (Danish Science Research Council) and LOMAPS (ESPRIT BRA 8130) projects.

References

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.

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

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

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.

Mark P. Jones. A theory of qualified types. In *ESOP '92*, volume 582 of *LNCS*, pages 287–306. Springer-Verlag, 1992.

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

Flemming Nielson and Hanne Riis Nielson. From CML to process algebras. In *CONCUR '93*, volume 715 of *LNCS*. Springer-Verlag, 1993.

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.

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.

Hanne Riis Nielson and Flemming Nielson. Static and dynamic processor allocation for higher-order concurrent languages. In *TAPSOFT'95 (FASE)*, volume 915 of *LNCS*, pages 590–604. Springer Verlag, 1995.

Hanne Riis Nielson and Flemming Nielson. Communication analysis for Concurrent ML. In *ML with Concurrency: Design, Analysis, Implementation and Application* (editor: Flemming Nielson), Springer-Verlag, December 1996.

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.

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

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

Geoffrey S. Smith. Polymorphic type inference with overloading and subtyping. In *TAPSOFT '93*, volume 668 of *LNCS*, pages 671–685. Springer-Verlag, 1993.

Yan-Mei Tang. *Systemes d'Effet et Interpretation Abstraite pour l'Analyse de Flot de Controle*. PhD thesis, Ecole Nationale Superieure des Mines de Paris, 1994. Report A/258/CRI. In English.

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

Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2), 1994.

A Output from Example 2.1

```
Type:
  ((a4 -b17-> a14) -b48-> (a4_list -b2-> a14_list))
Behaviour:
  e
Constraints:
  C: b5 > e
  C: b8 > r2_chan_a14_list
  C: b29 > e
  C: b18 > e
  C: b16 > e
  C: b28 > b26
  C: b26 > r2?a14_list
  C: b27 > e
  C: b57 > (b16;b17;b18;b27;b28;b29)
  C: b56 > fork_(b34)
  C: b55 > b42
  C: b42 > r2!a14_list
  C: b54 > e
  C: b53 > e
  C: b47 > e
  C: b51 > e
  C: b34 > (b47;b48;b51;b2;b53;b54;b55)
  C: b2 > (b5;(e+(b8;b56;b57)))
  C: b48 > e
```

B Proof of Proposition 6.2.

The proposition follows from the two lemmas below:

Lemma B.1

Suppose $E \vdash_2 e : t \,\&\, b$. Then also $\kappa(E) \vdash e : t \,\&\, b$. □

Proof

Induction in the proof tree. The only interesting case is "let":

Suppose that $E \vdash_2 \mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0 : t \,\&\, b$ because $E \vdash_2 e_1 : t_1 \,\&\, b_1$ and because $E \oplus [x : \forall \overline{F}.t_1] \vdash_2 e_0 : t \,\&\, b_0$ and because $b \sqsupseteq b_1; b_0$, where $F = \mathsf{fv}(E) \cup \mathsf{fv}(b_1)$.

By induction it holds that

$$\kappa(E) \ \vdash \ e_1 : t_1 \ \& \ b_1$$

and that $\kappa(E) \oplus [x : \forall\overline{F'}.t_1] \ \vdash \ e_0 : t \ \& \ b_0$; where $F' = F \cap \mathsf{fv}(t_1)$.
Let $F'' = (\mathsf{fv}(\kappa(E)) \cup \mathsf{fv}(b_1)) \cap \mathsf{fv}(t_1)$; then $F'' \subseteq F'$. Fact 4.9 then tells us that

$$\kappa(E) \oplus [x : \forall\overline{F''}.t_1] \ \vdash \ e_0 : t \ \& \ b_0$$

which is enough to show the desired judgement

$$\kappa(E) \ \vdash \ \mathsf{let} \ x{=}e_1 \ \mathbf{in} \ e_0 : t \ \& \ b.$$

Lemma B.2

Suppose $E \ \vdash \ e : t \ \& \ b$; and that $\kappa(E') = E$. Then also $E' \ \vdash_2 \ e : t \ \& \ b$. □

Before embarking on the proof, we first need an auxiliary concept: with $ts = \forall\overline{F}.t$ and $ts' = \forall\overline{F'}.t'$ type schemes, we say that $ts \stackrel{\alpha}{=} ts'$ (to be read "ts is alpha-equivalent to ts'") if and only if $ts' = ts[\psi]$ where ψ is a total bijective mapping from variables into variables such that $F \cap \mathsf{dom}(\psi) = \emptyset$ (so $F' = F$ and $F \cap \mathsf{ran}(\psi) = \emptyset$ and $t' = t[\psi]$). Clearly this is an equivalence relation. We say that $E \stackrel{\alpha}{=} E'$ holds if and only if $\mathsf{dom}(E) = \mathsf{dom}(E')$ and for all $x \in \mathsf{dom}(E)$ we have $E(x) \stackrel{\alpha}{=} E'(x)$ (so if $E \stackrel{\alpha}{=} E'$ then $\mathsf{fv}(E) = \mathsf{fv}(E')$). Some auxiliary results:

Fact B.3

Let ψ be a total *bijective* mapping from variables into variables. Then

(a) If $ts \succ t'$ then also $ts[\psi] \succ t'[\psi]$;
(b) if $ts \succ t'$ with ts a closed extended type scheme then also $ts \succ t'[\psi]$;
(c) $\mathsf{gen}(t_1, E, b_1)[\psi] = \mathsf{gen}(t_1[\psi], E[\psi], b_1[\psi])$.

Proof

Let ψ^{-1} be the inverse of ψ. For (a), write $ts = \forall\overline{F}.t$ such that $ts[\psi] = \forall\overline{F'}.t[\psi]$ with $F' = \mathsf{fv}(F[\psi])$. There exists an instance substitution ϕ with $\mathsf{dom}(\phi) \cap F = \emptyset$ such that $t' = t[\phi]$. Now define $\phi' = \psi^{-1}; \phi; \psi$; this ϕ' will suffice as instance substitution since

- $t[\psi][\phi'] = t[\phi][\psi] = t'[\psi]$;
- for $\gamma \in F'$ we have (as $\gamma[\psi^{-1}] \in F$) that $\gamma[\phi'] = \gamma[\psi^{-1}; \psi] = \gamma$, showing that $\mathsf{dom}(\phi') \cap F' = \emptyset$.

For (b), write $ts = \forall(t, C)$. There exists an instance substitution ϕ such that $t' = t[\phi]$ and such that $\phi \models C$. This shows that we have $ts \succ t'[\psi]$ as we can use $\phi; \psi$ as an instance substitution (since C contains C-constraints only so Fact 3.2 can be applied).

For (c), write $F = \mathsf{fv}(t_1) \cap (\mathsf{fv}(E) \cup \mathsf{fv}(b_1))$. The result then follows from the calculation

$$\begin{array}{ll} & \gamma \in \mathsf{fv}(F[\psi]) \\ \Leftrightarrow & \exists \gamma' \text{ such that } \gamma = \gamma'[\psi] \text{ and } \gamma' \in F \\ \Leftrightarrow & \gamma[\psi^{-1}] \in F \\ \Leftrightarrow & \gamma[\psi^{-1}] \in \mathsf{fv}(t_1) \text{ and } (\gamma[\psi^{-1}] \in \mathsf{fv}(E) \text{ or } \gamma[\psi^{-1}] \in \mathsf{fv}(b_1)) \\ \Leftrightarrow & \gamma \in \mathsf{fv}(t_1[\psi]) \text{ and } (\gamma \in \mathsf{fv}(E[\psi]) \text{ or } \gamma \in \mathsf{fv}(b_1[\psi])) \\ \Leftrightarrow & \gamma \in \mathsf{fv}(t_1[\psi]) \cap (\mathsf{fv}(E[\psi]) \cup \mathsf{fv}(b_1[\psi])) \end{array}$$

Fact B.4

Suppose $E \vdash e : t \;\&\; b$; and suppose that ψ is a total bijective mapping from variables into variables. Then also $E[\psi] \vdash e : t[\psi] \;\&\; b[\psi]$. □

Proof

A straight-forward induction in the proof tree, using Fact B.3 and Fact 3.2.

Fact B.5

Suppose that $E \stackrel{\alpha}{=} E'$ and $E \vdash e : t \;\&\; b$. Then also $E' \vdash e : t \;\&\; b$. □

Proof

Induction in the proof tree; the only interesting case is the base case $e = x$. Suppose $E \vdash x : t \;\&\; b$ because $E(x) \succ t$ and because $b \sqsupseteq \epsilon$. Let $E(x) = \forall \overline{F}.t_x$; there thus exists ϕ with $\mathsf{dom}(\phi) \cap F = \emptyset$ such that $t = t_x[\phi]$. We have $E'(x) = \forall \overline{F}.t'_x$ where $t'_x = t_x[\psi]$ with ψ a total bijective mapping from variables into variables such that $\mathsf{dom}(\psi) \cap F = \emptyset$.

Now define ϕ' as follows: if $\gamma \in \mathsf{fv}(t'_x)$ then $\gamma[\phi'] = \gamma[\psi^{-1}; \phi]$; and $\gamma[\phi'] = \gamma$ otherwise. It is clear that $t'_x[\phi'] = t_x[\phi] = t$ and that $\mathsf{dom}(\phi') \cap F = \emptyset$; which shows that $E' \vdash x : t \;\&\; b$. □

Now we are able to prove Lemma B.2:

Proof

Structural induction in e; there are two interesting cases:

$e = x$: Suppose $E \vdash x : t \;\&\; b$ because $E(x) \succ t$ and because $b \sqsupseteq \epsilon$. Let $E(x) = \forall \overline{F}.t_x$; then there exists a ϕ with $\mathsf{dom}(\phi) \cap F = \emptyset$ such that $t_x[\phi] = t$. We have $E'(x) = \forall \overline{F'}.t_x$, with $F' \cap \mathsf{fv}(t_x) = F$. Now let ϕ' behave as ϕ on $\mathsf{fv}(t_x)$ and as the identity otherwise; then $\mathsf{dom}(\phi') \cap F' = \emptyset$ and $t_x[\phi'] = t$. This shows that $E'(x) \succ t$ and hence $E' \vdash_2 x : t \;\&\; b$.

$e = \mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0$*:* Suppose $E \vdash \mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0 : t\ \&\ b$ because $E \vdash e_1 : t_1\ \&\ b_1$, because $E \oplus [x : \forall\overline{F}.t_1] \vdash e_0 : t\ \&\ b_0$ and because $b \sqsupseteq b_1; b_0$; with $F = \mathsf{fv}(t_1) \cap (\mathsf{fv}(E) \cup \mathsf{fv}(b_1))$.

Let $\vec{\gamma} = \mathsf{fv}(t_1) \setminus (\mathsf{fv}(E) \cup \mathsf{fv}(b_1))$; and let $\vec{\gamma}'$ be "fresh" copies of $\vec{\gamma}$. Let ψ be a substitution which maps $\vec{\gamma}$ into $\vec{\gamma}'$; which maps $\vec{\gamma}'$ into $\vec{\gamma}$ and which otherwise behaves as the identity. By Fact B.4 it holds (since $\mathsf{dom}(\psi) \cap \mathsf{fv}(b_1) = \emptyset$) that $E[\psi] \vdash e_1 : t_1[\psi]\ \&\ b_1$. It is easy to see (since $\mathsf{dom}(\psi) \cap \mathsf{fv}(E) = \emptyset$) that $E[\psi] \stackrel{\alpha}{=} E$ and hence we by Fact B.5 conclude that $E \vdash e_1 : t_1[\psi]\ \&\ b_1$. The induction hypothesis now tells us that

$$E' \vdash_2 e_1 : t_1[\psi]\ \&\ b_1. \tag{16}$$

Define $F' = \mathsf{fv}(E') \cup \mathsf{fv}(b_1)$. We have $F' \cap \mathsf{fv}(t_1[\psi]) = F' \cap (\mathsf{fv}(t_1) \setminus \vec{\gamma}) = F' \cap \mathsf{fv}(t_1) \cap (\mathsf{fv}(E) \cup \mathsf{fv}(b_1)) = \mathsf{fv}(t_1) \cap (\mathsf{fv}(E) \cup \mathsf{fv}(b_1)) = F$ which shows that

$$\kappa(E' \oplus [x : \forall\overline{F'}.t_1[\psi]]) = E \oplus [x : \forall\overline{F}.t_1[\psi]]. \tag{17}$$

Since $\mathsf{dom}(\psi) \cap F = \emptyset$ we conclude that $\forall\overline{F}.t_1 \stackrel{\alpha}{=} \forall\overline{F}.t_1[\psi]$. Fact B.5 now tells us that

$$E \oplus [x : \forall\overline{F}.t_1[\psi]] \vdash e_0 : t\ \&\ b_0. \tag{18}$$

Due to (17) we can apply the induction hypothesis on (18) to get

$$E' \oplus [x : \forall\overline{F'}.t_1[\psi]] \vdash_2 e_0 : t\ \&\ b_0 \tag{19}$$

and by combining (16) and (19) we arrive at the desired judgement

$$E' \vdash_2 \mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0 : t\ \&\ b.$$