Type and behaviour reconstruction for higher-order concurrent programs

TORBEN AMTOFT, FLEMMING NIELSON

and HANNE RIIS NIELSON

DAIMI, Aarhus University,
Ny Munkegade, DK-8000 Århus C, Denmark
(e-mail: {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.

Capsule Review

This paper is concerned with inferring behaviours for a skeletal language derived from Concurrent ML. A behaviour is a formula which expresses a communication pattern in terms of reading and writing on channels, sequencing, choice and recursion.

The paper improves on previous work by the authors by presenting a complete algorithm for behaviour inference. The algorithm reduces the problem of behaviour inference to a problem of constraint solving. The main result of the paper is that this reduction preserves all solutions: every possible behaviour annotation of a program can be obtained from that produced by the algorithm and a solvable constraint set. The main emphasis of the paper is on the generation of constraint sets, not the solution of constraint sets.

The soundness and completeness proofs of the algorithm are conducted with meticulous care. Besides the potential practical use of the analysis, the paper is valuable because of the algorithmic techniques and insights it contains.

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, 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, 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 kinds 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 the paper

Sections 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 sections 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 \( \text{let } x = e_1 \text{ in } e_0 \), conditionals \( \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \), recursive function definitions \( \text{rec } f(x) \Rightarrow e_0 \), 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 \( \text{map2} \) is a version of the well-known \( \text{map} \) 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 \( \text{channel} \); then \( \text{fork} \) creates a new process which computes \( \text{map2} f \text{ (tail } xs) \) and sends the result over the channel. The purpose of the constant \( \text{sync} \) is to convert a communication possibility into an actual communication (see Reppy (1991) for further motivation).

\[
\text{rec } \text{map2}(f) \Rightarrow \lambda xs.\text{if } x s = [] \text{ then } []
\]
\[
\text{else let } ch = \text{channel} ()
\]
\[
\text{in } \text{fork } (\lambda d.(\text{sync (send } \langle ch, \text{map2 } f \text{ (tail } xs) \rangle)));
\]
\[
\text{cons } (f \text{ (head } xs)) (\text{sync (receive } ch))
\]

The 'underlying type' of \( \text{map2} \) will be \((z_1 \rightarrow z_2) \rightarrow (z_1 \text{ list } \rightarrow z_2 \text{ list})\) and annotation with \textit{behaviour} information yields the type

\[
(z_1 \rightarrow^{b_1} z_2) \rightarrow^e (z_1 \text{ list } \rightarrow^{b_2} z_2 \text{ list})
\]

where \( b_2 \) (the behaviour of \( \text{map2} f \)) is expressed in terms of \( \beta_1 \) (the behaviour of \( f \)) as follows:

\[
\text{REC}_\beta ((e + ((z_2 \text{ list}) \text{CHAN}; \text{FORK } (\beta; !((x_2 \text{ list})); \beta_1; ?(x_2 \text{ list}))))).
\]

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 \( e \) denotes). Otherwise, it will first allocate a channel which transmits values of type \( z_2 \text{ list} \); 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 \( z_2 \text{ list} \); then it performs \( \beta_1 \) (by computing \( f \) on the head of the list); and finally it receives a value of type \( z_2 \text{ list} \).
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 $\alpha$, a base type like $\text{int}$ or $\text{bool}$ 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 $t \text{list}$, a communication type $t \text{com}$ (if such a communication possibility is
activated it behaves as indicated by $b$ and returns a value of type $t$), or a channel
type $t \text{chan}$ (a channel able to transmit values of type $t$).

Behaviours
Behaviours $b$ are built using the syntax

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

that is they can be one of the following: a behaviour variable $\beta$; the empty behaviour
$\epsilon$ (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 $t \text{CHAN}$ (a channel
able to transmit values of type $t$ is created); a fork action $\text{FORK} \;b$ (a process with
behaviour $b$ is created); a recursive behaviour $\text{REC} \beta \;b$ (where $b$ can ‘call’ itself
recursively via $\beta$); 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 $b = \text{REC} \beta \;b'$ binds $\beta$ in the sense that the set of free variables $fv(b)$ is
defined to be $fv(b') \setminus \{\beta\}$; 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.
In Figure 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 \([x : t]s\) is written \( E \oplus [x : ts]\); as usual type schemes take the form \( \forall \gamma.t \) where \( \gamma \) 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 \( \sqsubseteq \) on behaviours just as in Nielson and Nielson, 1994a, Table 3), with the intuitive interpretation that if \( b \sqsubseteq b' \) then \( b \) approximates \( b' \) in the sense that any action performed by \( b' \) can also be performed by \( b \). (To be more precise: \( \sqsubseteq \) is a subset of the simulation ordering which is undecidable, whereas \( \sqsubseteq \) 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 \( b \sqsubseteq b' \) then \( E \vdash e : t \& b' \).

The preorder is axiomatised in Figure 2, where \( b_1 \sqsupseteq b_2 \) denotes that \( b_1 \sqsupseteq b_2 \) and
P1 \( b \ni b \)

P2 \( b_1 \ni b_2 \land b_2 \ni b_3 \Rightarrow b_1 \ni b_3 \)

C1 \( b_1 \ni b_2 \land b_2 \ni b_4 \Rightarrow b_1 \ni b_2 \land b_2 \ni b_4 \)

C2 \( b_1 \ni b_2 \land b_3 \ni b_4 \Rightarrow b_1 + b_3 \ni b_2 + b_4 \)

C3 \( b_1 \ni b_2 \Rightarrow \text{FORK} b_1 \ni \text{FORK} b_2 \)

C4 \( b_1 \ni b_2 \Rightarrow \text{REC}\beta. b_1 \ni \text{REC}\beta. b_2 \)

S1 \( b_1; (b_2; b_3) = (b_1; b_2); b_3 \)

S2 \( (b_1 + b_2); b_3 = (b_1; b_3) + (b_2; b_3) \)

E1 \( b\equiv\epsilon; b \)

E2 \( b; \epsilon\equiv b \)

J1 \( b_1 + b_2 \ni b_1 \land b_1 + b_2 \ni b_2 \)

J2 \( b \ni b + b \)

R1 \( \text{REC}\beta. b \ni \beta(b \mapsto \text{REC}\beta. b) \)

\[ \ni \quad \text{Fig. 2. The preorder } \ni \text{ with equivalence } \equiv. \]

\( b_1 \ni b_1 \) (whereas ‘=’ denotes syntactic equality) and where \( b[\phi] \) denotes the result of applying the substitution \( \phi \) to \( b \). The axiomatisation expresses that ‘\( \ni \)’ is associative (S1) with \( \epsilon \) as neutral element (E1, E2); that ‘\( \ni \)’ is a congruence wrt. the various constructors (C1, C2, C3, C4); and that + is least upper bound wrt. \( \ni \) (J1, J2 together with C2 and P2).

Fact 3.2

If \( b_1 \ni b_2 \) then \( \text{fv}(b_1) \ni \text{fv}(b_2) \) and \( b_1[\phi] \ni b_2[\phi] \).

We now return to Figure 1. The clause for monomorphic abstraction says that if \( e_0 \) in an environment where \( x \) is bound to \( t_0 \) has type \( t_0 \) and behaviour \( b_0 \), then \( \lambda x.e_0 \) has type \( t_1 \ni b_0 t_0 \). 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 \( \epsilon \) (again reflecting that CML is call-by-value); as usual \( \forall \gamma. t \ni t' \) denotes that there exists \( \phi \) with \( \text{dom}(\phi) \subseteq \gamma \) such that \( t' = t[\phi] \). The clause for a polymorphic abstraction let \( x\equiv e_1 \text{ in } e_0 \) 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 \( \text{gen}(t_1, E, b_1) \):

\[ \text{gen}(t_1, E, b_1) = \forall \gamma. t_1 \text{ where } \gamma = \text{fv}(t_1) \setminus (\text{fv}(E) \cup \text{fv}(b_1)) \] (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 \( \text{CTypeOf}(c) \) which is an extended type scheme, that is of form \( \forall \gamma. (t, \Gamma) \) where \( \Gamma \) is a set of constraints of form \( b_1 \ni b_2 \) – such constraints are denoted \( C\text{-constraints} \) (for containment), and we say that a substitution \( \phi \) satisfies \( C \) if for all \( (b_1 \ni b_2) \in C \) it holds that \( b_1[\phi] \ni b_2[\phi] \).
Type and behaviour reconstruction

(according to Figure 2). Now \((\forall \gamma. (t, C)) > ' t\) holds iff there exists \(\phi\) with \(\text{dom}(\phi) \subseteq \gamma\) such that \(t' = t[\phi]\) and such that \(\phi\) satisfies \(C\). In addition we shall demand each \(\text{CTYPEO}(c)\) to be closed, that is \(\gamma = \text{fv}(t, C)\); accordingly we shall allow to write \(\forall (t, C)\) for \(\forall \gamma. (t, C)\) where \(\gamma = \text{fv}(t, C)\). Below we tabulate the value of \(\text{CTYPEO}()\) on some constants (all occurring in Example 2.1), this is adopted from (Nielson and Nielson, 1994b, Table 4):

\[
\begin{align*}
\text{head} : & \quad \forall (\alpha \text{ list} \rightarrow \beta, [\beta \supseteq e]) \\
\text{sync} : & \quad \forall ((\alpha \text{ com} \\beta_1) \rightarrow \beta_2, [\beta_2 \supseteq \beta_1]) \\
\text{send} : & \quad \forall (\alpha \text{ chan} \times \alpha \rightarrow^\beta \text{ com} \beta_2, [\beta_1 \supseteq e, \beta_2 \supseteq !\alpha]) \\
\text{receive} : & \quad \forall (\alpha \text{ chan} \rightarrow^\beta \text{ com} \beta_2, [\beta_1 \supseteq e, \beta_2 \supseteq ?\alpha]) \\
\text{channel} : & \quad \forall (\text{unit} \rightarrow^\beta \alpha, [\beta \supseteq \alpha \text{CHAN}]) \\
\text{fork} : & \quad \forall ((\text{unit} \rightarrow^\beta \alpha) \rightarrow^{\beta_2} \text{unit}, \beta_2 \supseteq \text{FORK} \beta_1)
\end{align*}
\]

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 \(b_1 \equiv b_2\); \(b'_1 \equiv b'\_2\) it does not necessarily hold that \(b_1 \equiv b'_1\) since we might have that \(b_1 = b'_1 = e\) and \(b_1 = b_2 = \text{!int}\).

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 for example, \(t_1 \rightarrow^b t_2\) is simple if and only if \(t_1\) and \(t_2\) are both simple and \(b = \beta\) for some \(\beta\)); 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 \(\text{REC} \beta. b\); a \(C\)-constraint is simple if it is of form \(\beta \supseteq b\) with \(b\) a simple behaviour; a substitution is simple if it maps type variables into simple
\[ \text{UNIFY}(\alpha, t) = \text{UNIFY}(t, \alpha) = [\alpha \mapsto t] \]

iff \( \alpha \notin \text{fv}(t) \) or \( t = \alpha \)

\[ \text{UNIFY}(\text{int}, \text{int}) = \text{UNIFY}(\text{bool}, \text{bool}) = \text{UNIFY}(\text{unit}, \text{unit}) = \text{id} \]

\[ \text{UNIFY}(\text{list}, t_2 \text{ list}) = \text{UNIFY}(t_1, \text{chan}, t_2, \text{chan}) = \emptyset \]

iff \( \text{UNIFY}(t_1, t_2) = \emptyset \)

\[ \text{UNIFY}(\text{com} \beta_1, t_2 \text{ com} \beta_2) = 0'; [\beta'_1 \mapsto \beta'_2] \]

iff \( \text{UNIFY}(t_1, t_2) = 0' \) and \( \beta'_1 = \beta_1[0'] \)

\[ \text{UNIFY}(t_1 \times t_2, t_1 \times t_2) = 0'; 0'' \]

iff \( \text{UNIFY}(t_1, t_2) = 0' \)

and \( \text{UNIFY}(t_1[0'], t_2[0']) = 0'' \)

\[ \text{UNIFY}(t_1 \rightarrow^\beta_1 t_1, t_2 \rightarrow^\beta_2 t_2) = 0'; 0''; [\beta'_1 \mapsto \beta'_2] \]

iff \( \text{UNIFY}(t_1, t_2) = 0' \)

and \( \text{UNIFY}(t_1[0'], t_2[0']) = 0'' \)

and \( \beta'_1 = \beta_1[0'; 0''] \)

\[ \text{UNIFY}(t_1, t_2) \text{ fails otherwise} \]

Fig. 3. Procedure UNIFY.

types and maps behaviour variables into behaviour variables (rather than simple behaviours).

**Fact 4.1**
Simple types are closed under the application of simple substitutions: \( t[\phi] \) is simple if \( t \) and \( \phi \) are; similarly for behaviours and C-constraints. Also simple substitutions are closed under composition: \( \phi; \phi' \) (first \( \phi \) and then \( \phi' \)) is simple if both \( \phi \) and \( \phi' \) are.

**Fact 4.2**
All CTypeOf(c) have simple types and simple C-constraints.

In Figure 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 UNIFY(t₁, t₂) = θ with t₁ and t₂ simple. Then t₁[θ] = t₂[θ]. □

Proof
Induction in the definition of UNIFY, using the same terminology. For the call UNIFY(θ, t) we have θ[θ] = t[θ] where we employ that θ ̸∈ fv(t) (or t = θ).

Next suppose UNIFY(t₁ com β₁, t₂ com β₂) = θ with θ = θ′; [β₁′ → β₂′]. By induction we have t₁[θ′] = t₂[θ′], which implies t₁[θ] = t₂[θ] and hence

\[(t₁ com β₁)[θ] = t₁[θ] com β₁′[β₁′ → β₂′] = t₂[θ] com β₂′ = (t₂ com β₂)[θ].\]

The remaining cases are similar. □

Lemma 4.5
Suppose t₁[ψ] = t₂[ψ] with t₁ and t₂ simple. Then UNIFY(t₁, t₂) succeeds with result θ, and there exists ψ′ such that ψ = θ; ψ′. □

Proof
Induction in the sum of the sizes of t₁ and t₂. If one of these is a variable, then the claim follows from the fact that if x[ψ] = t[ψ] then x ̸∈ fv(t) or x = t, and also ψ = [x → t]; ψ.

Otherwise, they must have the same topmost constructor say com (the other cases are rather similar). That is, the situation is that (t₁ com β₁)[ψ] = (t₂ com β₂)[ψ]. Since t₁[ψ] = t₂[ψ] we can apply the induction hypothesis to infer that the call UNIFY(t₁, t₂) succeeds with result θ′ and that there exists ψ′ such that ψ = θ′; ψ′.

With β₁′ = β₁[θ′], with β₂′ = β₂[θ′] and with θ = θ′; [β₁′ → β₂′] we conclude that UNIFY(t₁ com β₁, t₂ com β₂) succeeds with result θ. Since β₁′[ψ′] = β₁[θ′; ψ] = β₂[θ′; ψ] = β₂′[ψ′] it holds that ψ′ = [β₁′ → β₂′]; ψ′. Hence we have the desired relation ψ = θ′; ψ′ = θ; ψ′. □

4.2 A previous approach with S-constraints

A (sound but not complete) reconstruction algorithm for the inference system in Figure 1 was presented in Nielson and Nielson (1994b). Inspired by, for instance, 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 let x = e₁ in e₀, where x occurs twice in e₀; here e₀ must be analysed in an environment where x is bound to an extended type scheme ∀γ.⟨t, C⟩ with C the constraints generated when analysing e₁. The first occurrence of x in e₀ gives rise to a copy of C (and t), where the polymorphic variables γ are replaced by fresh variables γ ′. Let C₀ be the constraints generated when analysing e₀; it is easy to see that given γ₀ satisfying C₀ we can find a substitution γ′ which satisfies C by stipulating that γ[γ′] = γ[γ₀] and that γ′ equals γ₀ on fv(γ). With γ″ the fresh variables generated by the second occurrence of x in e₀, we can find yet another substitution γ″ satisfying C (now we stipulate γ[γ″] = γ″[γ₀]).
The problem is that $\psi'$ and $\psi''$ 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 $(\beta \sqsupseteq !\text{int}; !\text{int}; \beta)$. Both the substitution $\psi_1$ which maps $\beta$ into $\text{REC}\beta.(\text{int}; !\text{int}; \beta)$ and the substitution $\psi_2$ which maps $\beta$ into $\text{REC}\beta.(!\text{int}; \beta)$ will satisfy this constraint; but with the current axiomatisation it seems hard to find a sense in which $\psi_1$ and $\psi_2$ 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 $\text{let } x = e_1 \text{ in } e_0$, 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 $\psi$ 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 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 $\psi$ must satisfy that for every polymorphic variable $\gamma$ and every non-polymorphic variable $\gamma'$, the sets $\text{fv}(\gamma[\psi])$ and $\text{fv}(\gamma'[\psi])$ 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:

**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 $\text{fv}(b) \subseteq F$.

**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 $\text{fv}(b) \cap F \neq \emptyset$ then $\beta \in F$.

We define the downwards closure of $F$ wrt. $C$, denoted $F \downarrow C$, as the least set which

† A remedy might be to adopt more rules for behaviours such that $\text{REC}\beta.b$ is equivalent to its infinite unfolding (cf. rule R1 in Figure 2 which states that $\text{REC}\beta.b$ is equivalent to its finite unfoldings, and cf. Cardone and Coppo (1991), where a similar change in axiomatisation is made concerning recursive types).
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 $\mathcal{N}$ denotes the set of variables not quantified over. Demanding $\mathcal{N}$ to be downwards closed amounts to stating that a non-polymorphic variable cannot have polymorphic subparts (which seems reasonable); whereas additionally demanding $\mathcal{N}$ 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 $\mathcal{N}$ 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 $\mathcal{N}$ 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 $(\forall F.\vec{g} \succ \vec{g}')$; 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 $(\forall F.\vec{g} \succ \vec{g}')$ is satisfied by $\psi$ if and only if there exists an “instance substitution” $\phi$, with $\text{dom}(\phi)$ disjoint from $\text{fv}(F[\psi])$, 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 $(\forall F.\vec{\gamma} \succ \vec{\gamma}')$ is generated where $\vec{\gamma}'$ are fresh copies of $\vec{\gamma}$ and where $F = \text{fv}(t,C) \setminus \vec{\gamma}$; 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 $\text{fv}(C)$ 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 $\forall F.t$ 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 $\text{fv}(ts) = F$; and say that $ts$ is simple if $t$ is.
There is a natural injection from type schemes in the classical form $\forall \bar{\gamma}.t$ into type schemes in the new form (let $F = \text{fv}(t) \setminus \bar{\gamma}$). A type scheme $\forall \bar{F}.t$ which is in the image of this injection (i.e. where $F \subseteq \text{fv}(t)$) 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: $\forall \bar{F}.t \succ t'$ holds if and only if there exists an ‘instance substitution’ $\phi$ with $\text{dom}(\phi) \cap F = \emptyset$ such that $t' = t[\phi]$.

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

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

We still need extended type schemes of form $\forall \bar{F}.(t, C)$, with $C$ being C-constraints, to appear in $\text{CTypeOf}(c)$; 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 $\phi$ such that $\phi$ satisfies $C$ and $t' = t[\phi]$.

Thus the inference system in Figure 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 $ts = \forall \bar{F}.t$ and $ts' = \forall \bar{F}'.t'$ 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 $E' \vdash e : t \& b$. \(\Box\)

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

**Definition 4.10**

If $ts = \forall \bar{F}.t$ then $ts[y] = \forall \bar{F}.t[y]$ where $F' = \text{fv}(F[y])$.

The result of applying $\psi$ to the S-constraint $\forall \bar{F}.\bar{g} > \bar{g}'$ is $\forall \bar{F}.\bar{g}[y] > \bar{g}'[y]$ where again $F' = \text{fv}(F[y])$. \(\Box\)

Notice that the S-constraint $\forall \bar{F}.t \succ t'$ is satisfied by $\psi$ (cf. Def. 4.8) if and only if the type $t'[y]$ is an instance of the (enriched) type scheme $(\forall \bar{F}.t)[y]$. In the following

\$ One should point out that in general it no longer holds that if $ts > t$ then $ts[y] > t[y]$ (consider, for example, $ts = \forall \bar{x}_1 \times \bar{x}_2. t = \text{int} \times \bar{x}_2$ and $\psi = [\bar{x}_2 \mapsto \bar{x}_1 \times \bar{x}_2]$, where the ‘bound’ variable $\bar{x}_1$ occurs in the range of $\psi$). 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).
we shall often write \( \psi \models C \) to denote that \( \psi \) 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 \( \psi \) and \( \psi' \) be substitutions. Then

- \( \text{fv}(\text{fv}(ts[\psi])) = \text{fv}(ts[\psi]) \);
- \( ts[\psi; \psi'] = ts[\psi][\psi'] \);
- \( \psi; \psi' \models C \) 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 Figures 4 and 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 let \( \textit{x := e_1 \textbf{ in } ...x ...x } \) 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 \( ...c ...c ...c ...c ...c \) the types of the two \( c \)'s must be instances of a type that we know already.

**Properties of \( \mathcal{N} \) \( \mathcal{B} \)**

In Figure 5 we defined

\[
\mathcal{N} \mathcal{B} = \mathcal{E} \mathcal{B} \downarrow C_1
\]

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

\[
\begin{align*}
\psi \models C_1 & \Rightarrow \text{fv}(\mathcal{N} \mathcal{B}[\psi]) \supseteq \text{fv}(\mathcal{E} \mathcal{B}[\psi]) & (3) \\
\psi \models C_1 & \Rightarrow \text{fv}(\mathcal{N} \mathcal{B}[\psi]) \subseteq \text{fv}(\mathcal{E} \mathcal{B}[\psi]) & (4)
\end{align*}
\]

It turns out that to prove soundness of \( W \), all we need to know about \( \mathcal{N} \mathcal{B} \) is that it satisfies (3); and it turns out that to prove completeness of \( W \), all we need to know about \( \mathcal{N} \mathcal{B} \) 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 Figure 1.

**Theorem 5.1**
\[ W(x, E) = (t, b, C, \theta) \]
if \( E(x) = \forall \mathcal{T}_x.t_x \) and \( \bar{\gamma} = \mathcal{T}(t_x) \backslash F_x \) and \( \bar{\gamma}' \) are fresh copies of \( \bar{\gamma} \)
and \( t = t, [\bar{\gamma} \mapsto \bar{\gamma}'] \) and \( b = \epsilon \) and \( C = \forall \mathcal{T}_x.\bar{\gamma} > \bar{\gamma}' \) and \( \theta = \text{id} \)

\[ W(c, E) = (t, b, C, \theta) \]
if \( \text{CTYPEOf}(c) = \forall (t_x, C_i) \)
and \( \bar{\gamma} = \mathcal{T}(t_x) \cup \mathcal{T}(C_i) \) and \( \bar{\gamma}' \) are fresh copies of \( \bar{\gamma} \)
and \( t = t, [\bar{\gamma} \mapsto \bar{\gamma}'] \) and \( b = \epsilon \) and \( C = C_i[\bar{\gamma} \mapsto \bar{\gamma}'] \) and \( \theta = \text{id} \)

\[ W(\lambda x.e_0, E) = (t, b, C, \theta) \]
if \( \alpha_1 \) is a fresh variable and \( W(e_0, E \oplus [x : \alpha_1]) = (t_0, b_0, C_0, \theta_0) \)
and \( \beta_0 \) is a fresh variable and \( t = \alpha_1[\theta_0] \) and \( b = \beta_0 \) and \( C = C_0 \oplus \beta_0 \) and \( \theta = \theta_0 \)

Fig. 4. Algorithm \( W \), first part.

\[ W(\text{let } x = e_1, \text{ in } e_0, E) = (t, b, C, \theta) \]
if \( W(e_1, E) = (t_1, b_1, C_1, \theta_1) \)
and \( W(e_0, E[\theta_1]) \oplus [x : \forall \mathcal{T}_x t_x] = (t_0, b_0, C_0, \theta_0) \)
and \( t = t_0 \) and \( b = \beta_1 \oplus b_0 \) and \( C = C_1 \oplus C_0 \) and \( \theta = \theta_1 \oplus \theta_0 \)
where \( \delta' = \mathcal{T}(E[\theta_1]) \cup \mathcal{T}(b_1) \) and \( \chi = \delta \delta' \chi' \)

\[ W(\text{if } e_0 \text{ then } e_1 \text{ else } e_2, E) = (t, b, C, \theta) \]
if \( W(e_0, E) = (t_0, b_0, C_0, \theta_0) \)
and \( W(e_1, E[\theta_0]) = (t_1, b_1, C_1, \theta_1) \)
and \( W(e_2, E[\theta_0]) = (t_2, b_2, C_2, \theta_2) \)
and \( \text{UNIFY}(t_0[\theta_1], t_1[\theta_2] \times t_2[\theta_1], \theta_0 \times \theta_2) = 0' \)
and \( t = t_2[\theta_0] \) and \( b = (b_0[\theta_1]; b_1[\theta_2] + b_2[\theta_1])[\theta_0'] \)
and \( C = (C_0[\theta_1]; C_1[\theta_2]; C_2[\theta_1])[\theta_0'] \) and \( \theta = \theta_0; \theta_1; \theta_2; \theta_0' \)

\[ W(\text{rec } f(x) \Rightarrow e_0, E) = (t, b, C, \theta) \]
if \( \alpha_1, \alpha_2, \beta \) are fresh variables
and \( W(e_0, E \oplus [f : \alpha_1 \mapsto z_2] \oplus [x : \alpha_1]) = (t_0, b_0, C_0, \theta_0) \)
and \( \text{UNIFY}(z_2[\theta_0]; \theta_0) = 0' \)
and \( t = (z_2[\theta_0] \mapsto \beta[\theta_0]) \) and \( b = \epsilon \)
and \( C = (C_0[\beta[\theta_0]; b_0])[\theta_0'] \) and \( \theta = \theta_0; \theta_0' \)

Fig. 5. Algorithm \( W \), second part.
Suppose that $W(e, \emptyset) = (t, b, C, \emptyset)$ and that $\varphi$ is such that $\varphi \models C$ and $t' = \iota[\varphi]$ and $b' \supseteq b[\varphi]$. Then it holds that 

\[ \emptyset \vdash e : t' \land b'. \]

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 $ts = \forall F.t$ we define $\kappa(ts) = \forall F'.t$ where $F' = F \cap \text{fv}(t)$.

**Proposition 5.2**

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

**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 if $e_0$ then $e_1$ else $e_2$, 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 $F' = \text{fv}(F, x[\varphi])$ and let $F'' = F' \cap \text{fv}(t_0[\varphi])$.

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

\[ \forall F''. (t_0[\varphi]) \succ t_1[[\varphi] \mapsto \gamma'][\varphi]. \] (5)

Since $\varphi \models C$ we have $\forall F'. \gamma'[\varphi] \succ \gamma'[\varphi]$; so there exists a $\phi'$ with $\text{dom}(\phi') \cap F' = \emptyset$ such that $\varphi; \phi' \models [\varphi] \mapsto \gamma'; \psi$ on $\gamma$. This implies, since $\text{fv}(t_0) \subseteq \gamma \cup F_0$, that we even have that $\varphi; \phi' \models [\varphi] \mapsto \gamma'; \psi$ on $\text{fv}(t_0)$. But this shows that (5) holds, with $\phi'$ as the ‘instance substitution’.

**The case $W(c, E)$**: Since $\varphi \models C$ we have $[\varphi] \mapsto \gamma'; \psi \models C_2$ so it holds that $\forall (t_1, C_2) \succ t_1[[\varphi] \mapsto \gamma'; \psi]$. Therefore we have the inference

\[ \kappa(E[\emptyset][\varphi]) \vdash c : t_1[[\varphi] \mapsto \gamma'; \psi] \land e \]

which amounts to the desired relation.

**The case $W(e_1, e_2, E)$**: Since $\varphi \models C$ 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

\[ \kappa(E[\emptyset][\varphi]) \vdash e_1 : t_1[\theta_2 ; \theta_0 ; \psi] \land b_1[\theta_2 ; \theta_0 ; \psi]. \]

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

\[ \kappa(E[\emptyset][\varphi]) \vdash e_1 : t_2[\theta_0 ; \psi] \rightarrow \beta_0[\theta_0 ; \psi] \land \beta_1[\theta_0 ; \theta_0 ; \psi]. \]

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

\[ \kappa(E[\emptyset][\varphi]) \vdash e_2 : t_2[\theta_0 ; \psi] \land b_2[\theta_0 ; \psi]. \]
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
\[ \kappa(E[\theta][\psi]) \otimes [x : z_1[\theta_0][\psi]] \vdash e_0 : t_0[\psi] \& b_0[\psi]. \]
By Fact 3.1 and the fact that $\beta_0[\psi] \supseteq b_0[\psi]$, we therefore have
\[ \kappa(E[\theta_0][\psi]) \otimes [x : z_1[\theta_0][\psi]] \vdash e_0 : t_0[\psi] \& \beta_0[\psi]. \]
This shows the desired judgement
\[ \kappa(E[\theta_0][\psi]) \vdash \lambda x.e_0 : (z_1[\theta_0] \rightarrow \beta_0) t_0[\psi] \& e. \]

The case $W(\text{let } x=e_1 \text{ in } e_0, E)$ 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
\[ \kappa(E[\theta][\psi]) \vdash e_1 : t_1[\theta_0; \psi] \& b_1[\theta_0; \psi]. \]
To arrive at the desired judgement
\[ \kappa(E[\theta][\psi]) \vdash \text{let } x=e_1 \text{ in } e_0 : t_0[\psi] \& b[\psi] \]
we must show that
\[ \kappa(E[\theta][\psi]) \otimes [x : \forall F'' t_1[\theta_0; \psi]] \vdash e_0 : t_0[\psi] \& b_0[\psi] \quad (6) \]
where $F'' = (\text{fv}(\kappa(E[\theta][\psi])) \cup \text{fv}(b_1[\theta_0; \psi])) \cap \text{fv}(t_1[\theta_0; \psi])$.

Since $\psi \models C_0$ we can apply the induction hypothesis on the call $W(e_0, E[\theta_1] \otimes [x : \ldots])$ and the substitution $\psi$ to get
\[ \kappa(E[\theta][\psi]) \otimes [x : \forall F' t_1[\theta_0; \psi]] \vdash e_0 : t_0[\psi] \& b_0[\psi] \quad (7) \]
where $F' = \text{fv}(\forall \forall^2[\theta_0; \psi]) \cap \text{fv}(t_1[\theta_0; \psi])$.

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 $C_1$):
\[
\text{fv}(\kappa(E[\theta][\psi])) \cup \text{fv}(b_1[\theta_0; \psi]) \\
\subseteq \text{fv}(E[\theta][\psi]) \cup \text{fv}(b_1[\theta_0; \psi]) = \text{fv}((\text{fv}(E[\theta_1]) \cup \text{fv}(b_1))[\theta_0; \psi]) = \text{fv}(E[\theta][\psi]) \\
\subseteq \text{fv}(\forall \forall^2[\theta_0; \psi]).
\]
This concludes the proof of Proposition 5.2. \( \square \)

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:
Suppose Proposition 6.3. Then $\emptyset \vdash f_{\text{rec}}$. Hence inferences in this system will be of form $\vdash e : t \& b$. Assume Proposition 6.2. So to prove Theorem 6.1, it will be sufficient to show Proposition 6.3 below that $\emptyset \vdash e : t \& b$. (In particular, $\emptyset \vdash e : t \& b$ holds if and only if $\emptyset \vdash e : t \& b$.)

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 Figure 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 $\text{gen}(t_1, E, b_1) = \forall T. t_1$ where $F = \text{fv}(t_1) \cap (\text{fv}(E) \cup \text{fv}(b_1))$; this is in contrast to the algorithm where no intersection with $\text{fv}(t_1)$ 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:

$$\text{gen}_2(t_1, E, b_1) = \forall T. t_1$$

Hence inferences in this system will be of form $E \vdash e : t \& b$ 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 $E' \vdash e : t \& b$. (In particular, $\emptyset \vdash e : t \& b$ holds if and only if $\emptyset \vdash e : t \& b$.)

In the formulation of Proposition 6.3 we write $\phi_1 \equiv E \phi_2$ to denote that $\forall \gamma \in \text{var}(E)$ where $\text{var}(E)$ contains all the variables occurring in $E$; that is, $\text{var}(E)$ is the union over $\text{dom}(E)$ of $\text{var}(E(x))$ where $\text{var}(T) = T \cup F$. For a substitution $\phi$ we define $\text{var}(\phi) = \text{fv}(\phi) = \text{dom}(\phi) \cup \text{ran}(\phi)$.

Proposition 6.3. Suppose $E' \vdash e : t \& b'$ and $E[\phi] \geq E'$ with $E$ simple. Then $W(e, E)$ succeeds with result $(t, b, C, \emptyset)$ (all simple) and there exists a $\psi$ such that $\emptyset ; \psi \equiv E \phi$ and $\emptyset \vdash C$ and $t' = t[\psi]$ and $b' \supseteq b[\psi]$. (Note that $E$ 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.)

The proof is by induction on the proof tree for $E' \vdash e : t \& b'$; 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 if $e_0$ then $e_1$ else $e_2$, $\text{rec } f(x) \Rightarrow e_0$, 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 \( E'(x) = \forall F_x \tau_c \), because \( t' = t'_c(\phi') \) with \( \text{dom}(\phi') \cap F_x = \emptyset \), and because \( b' \supseteq e \).

Since \( E[\phi] \supseteq E' \) it holds that \( t_c[\phi] = t'_c \) (thanks to our syntactic definition of \( \supseteq \)) and that \( \text{fv}(F_c[\phi]) \subseteq F'_c \).

From this we infer that
\[
\text{dom}(\phi') \cap \text{fv}(F_c[\phi]) = \emptyset. \tag{9}
\]

Now define \( \psi \) as follows: it maps \( \vec{\gamma}' \) into \( \vec{\gamma}[\phi'; \phi] \); and otherwise it behaves like \( \phi \). This ensures that \( \theta; \psi \triangleq \phi \) and it is trivial that \( b' \supseteq \psi[\phi] \).

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 \( \psi \) equals \( \phi \) on \( F_x \cup \vec{\gamma} \) this implies \( \forall \vec{F}_x[\psi].\vec{\gamma}[\psi] > \vec{\gamma}[\psi] \) (with \( \phi' \) as the instance substitution) which amounts to \( \psi \models C \). Finally, since \( \text{fv}(t_c) \subseteq F_x \cup \vec{\gamma} \) we also get from (10) that \( t' = t'_c[\phi'] = t_c[\phi; \phi'] = t_c[\vec{\gamma} \mapsto \vec{\gamma}]; \psi = t[\psi] \).

The case \( e = c \) Let \( \text{CTYPEOf}(c) = \forall (t_c, C_c) \). Suppose \( E' \vdash_2 c : t' \& b' \) holds because \( b' \supseteq e \) and because there exists a \( \psi' \) with \( \psi' \models C_c \) such that \( t' = t_c[\psi'] \). Now define \( \psi \) as follows: it maps \( \vec{\gamma}' \) into \( \vec{\gamma}[\psi'] \); and otherwise it behaves like \( \phi \). 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 \( \theta; \psi \triangleq \phi \) and that \( b' \supseteq b[\psi] \).

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 \( t'_1 = t'_2 \supseteq b'_0 \) and \( t' = t'_1 \).

By induction, we see that the call \( W(e_1, E) \) succeeds and that there exists \( \psi_1 \) such that \( \theta_1; \psi_1 \triangleq \phi \), such that \( \psi_1 \models C_1 \), such that \( t'_1 = t_1[\psi_1] \), and such that \( b'_1 \supseteq b[\psi_1] \).

Since \( E[\theta_1][\psi_1] = E[\phi] \) we infer that \( E[\theta_1][\psi_1] \supseteq E' \). 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 \( \psi_2 \) such that \( \theta_2; \psi_2 \) equals \( \psi_1 \) on \( \text{var}(E[\theta_1]) \), such that \( \psi_2 \models C_2 \), such that \( t'_2 = t_2[\psi_2] \), and such that \( b'_2 \supseteq b[\psi_2] \).

Some terminology: let \( V_1 = \text{var}(t_1, b_1, C_1, \theta_1) \) and \( V_2 = \text{var}(t_2, b_2, C_2, \theta_2) \) and \( E_1 = \text{var}(E[\theta_1]) \); 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 \( \text{dom}(\theta_2) \) or in \( \text{ran}(\theta_2) \).

Now define \( \psi_0 \) to behave as \( \psi_2 \) except that it behaves as \( \psi_1 \) on internal variables; it maps \( x \) into \( t' \); and it maps \( \beta_0 \) into \( b'_0 \).

We have the following relations:
\[
\psi_0 \triangleq \psi_2 \text{ and } \theta_2; \psi_0 \triangleright E_1 \triangleright \psi_1 \tag{11}
\]
where the second part follows from the following reasoning: if \( \gamma \) is internal then \( \gamma[\theta_2; \psi_0] = \gamma[\psi_0] = \gamma[\psi_1] \); and if \( \gamma \) 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
\[ t_1[\theta_2][\psi_0] = t_1[\psi_1] = t'_1 = t'_2 \rightarrow^{b_0} t' = (t_2 \rightarrow^{\rho_b} z)[\psi_0] \]
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 \( \psi \) such that \( \psi_0 = \theta_0 ; \psi \). Using this and (11) we can infer the desired properties of \( \psi \):

- If \( \gamma \in \text{var}(E) \) 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 \( \psi_2 \models C_2 \).

- \( t' = z[\psi_0] = z[\theta_0; \psi] = t[\psi] \).

- Since \( \sqsubseteq \) is a pre-congruence (Rule C1 in Figure 2) we infer that
\[ b' \sqsubseteq b'; b'_0 \sqsubseteq b_1[\psi_1]; b_2[\psi_2]; b_0[\psi_0] = b_1[\theta_2][\psi_0]; b_2[\psi_0]; b_0[\psi_0] = b[\psi]. \]

The case \( e = \lambda x.e_0 \) Suppose \( E' \vdash_2 \lambda x.e_0 : t' \sqsubseteq b' \) holds because we with \( t' = t'_1 \rightarrow^{b_0} t'_0 \) and \( b' \sqsubseteq e \) have \( E' \oplus [x : t'_1] \vdash_2 e_0 : t'_0 \) and \( b'_0 \). Define \( \phi_0 \) to behave like \( \phi \) except that it maps \( \alpha \) as follows: it maps \( \alpha \) into \( t'_1 \). Then we clearly have
\[ (E \oplus [x : \alpha])(\phi_0) \geq E' \oplus [x : t'_1] \]
so by induction we see that the call \( W(e_0, E \oplus [x : \alpha]) \) succeeds and that there exists \( \psi_0 \) such that \( \theta_0 ; \psi_0 \models \phi_0 \); such that \( \psi_0 \models C_0 \); such that \( t'_0 = t_0[\psi_0] \) and such that \( b'_0 \sqsubseteq b_0[\psi_0] \).

Define \( \psi \) as follows: it maps \( \beta_0 \) into \( b'_0 \); and otherwise it behaves like \( \psi_0 \). It is obvious that \( \theta ; \psi \models E \) and that \( \psi \models C_0 \). Since it moreover holds that
\[ b_0[\psi] = b'_0 \sqsubseteq b_0[\psi] \]
we conclude that \( \psi \models C \). Clearly \( b' \sqsubseteq b[\psi] \), and finally we have
\[ t' = t'_1 \rightarrow^{b_0} t'_0 = z[\phi_0] \rightarrow^{b_0[\psi]} = (z[\theta_0] \rightarrow^{\rho_b} t_0)[\psi]. \]

The case \( e = \text{let } x = e_1 \text{ in } e_0 \) Suppose \( E' \vdash_2 \text{let } x = e_1 \text{ in } e_0 : t' \sqsubseteq b' \) because of \( E' \vdash_2 e_1 : t'_1 \) and \( b'_1 \) and of \( F' = \text{fv}(E') \cup \text{fv}(b'_1) \) and of \( E' \oplus [x : \forall \mathcal{T}. t'_1] \vdash_2 e_0 : t' \sqsubseteq b'_0 \) and because of \( b' \sqsubseteq b'_1 \) \( b'_0 \). By induction we see that \( W(e_1, E) \) succeeds and that there exists \( \psi_1 \) such that \( \theta_1 ; \psi_1 \models E \), such that \( \psi_1 \models C_1 \), such that \( t'_1 = t_1[\psi_1] \), and such that \( b'_1 \sqsubseteq b_1[\psi_1] \).

To apply the induction hypothesis once more we must show
\[ (E[\theta_1] \oplus [x : \forall \mathcal{T}. t_1])[\psi_1] \geq E' \oplus [x : \forall \mathcal{F}. t'_1]. \]

But this is an easy consequence of the calculation
\[
\begin{align*}
\text{fv}(\forall \mathcal{F}. t_1) & \subseteq \text{fv}(\text{fv}(E[\theta_1]) \cup \text{fv}(b_1))[\psi_1]) \\
& = \text{fv}(E[\phi]) \cup \text{fv}(b_1[\psi_1]) \\
& \subseteq \text{fv}(E') \cup \text{fv}(b_1[\psi_1]) \\
& \subseteq \text{fv}(E') \cup \text{fv}(b_1) = F'
\end{align*}
\]
where the first inclusion follows from (4) (cf. section 4.6) used on \( \psi_1 \); 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 \( \psi_0 \) such that \( \theta_0; \psi_0 \) equals \( \psi_1 \) on \( V_e \) where \( V_e = \text{var}(E[\theta_1]) \cup \mathcal{N}^2 \cup \text{fv}(t_1) \), such that \( \psi_0 \models C_0 \), such that \( t' = t_0[\psi_0] \) and such that \( b'_0 \triangleright b_0[\psi_0] \).

Then some terminology (similar to that introduced in the case for application): let \( V_1 = \text{var}(t_1, b_1, C_1, \theta_1) \) and \( V_0 = \text{var}(t_0, b_0, C_0, \theta_0) \); 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 \( \psi \) to behave as \( \psi_0 \), except that is behaves as \( \psi_1 \) on internal variables. We have the following relations:

\[
\psi \overset{V_e}{=} \psi_0 \text{ and } \theta_0; \psi \overset{V_e \cup V}{=} \psi_1
\]  

where the second part follows from the following reasoning: if \( \gamma \) is internal then \( \gamma[\theta_0; \psi] = \gamma[\psi] = \gamma[\psi_1] \); and if \( \gamma \) 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 \( \psi \): (i) if \( \gamma \in \text{var}(E) \) 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 \( b' \triangleright b'_1 \); \( b_0 \triangleright b_0[\psi_1] \); \( b_0[\psi_0] = b_1[\theta_0][\psi] \); \( b_0[\psi] = b[\psi] \). \( \square \)

### 7 Choice of generalisation strategy

Recall that in Figure 5 we defined \( \mathcal{N}^2 \) as \( \varepsilon \mathcal{B} \cap C_1 \) and we saw that to prove soundness it is sufficient to know that \( \mathcal{N}^2 \) satisfies (3):

\[
\psi \models C_1 \Rightarrow \text{fv}(\mathcal{N}^2[\psi]) \supseteq \text{fv}(\varepsilon \mathcal{B}[\psi])
\]

and to prove completeness it is sufficient to know that \( \mathcal{N}^2 \) satisfies (4):

\[
\psi \models C_1 \Rightarrow \text{fv}(\mathcal{N}^2[\psi]) \subseteq \text{fv}(\varepsilon \mathcal{B}[\psi]).
\]

In this section we shall investigate whether other definitions of \( \mathcal{N}^2 \) might be appropriate.

**Requiring \( \mathcal{N}^2 \) to be upwards closed?** (In addition to being downwards closed and contain \( \varepsilon \mathcal{B} \); 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:

\[
\lambda x. \text{let } f = \lambda y. \text{let } \text{ch1} = \text{channel} (\_ ) \text{ in } \text{let } \text{ch2} = \text{channel} (\_ )
\]

\[
\text{ in } \lambda h.(\text{sync (send <ch1,x>)));
\]

\[
\text{ (sync (send <ch2,y>))})
\]

\[
; \ y
\]

\[
\text{in } f \ 7; \ f \ \text{true}
\]
which is typable since with \( E = \{ x : z_x \} \) we have
\[
E \vdash \lambda y_\ldots : z_y \rightarrow z_x; \text{CHAN}; z_y; \text{CHAN} \quad \alpha y \& \epsilon
\]
and hence it is possible to quantify over \( z_y \). On the other hand, when analysing \( \lambda y_\ldots \)
the algorithm will generate constraints whose ‘transitive closure’ includes something like
\[
\beta \sqsupseteq x; \text{CHAN}; z_y; \text{CHAN}
\]
and since \( x \) belongs to \( \mathcal{EB} \) and hence to \( \mathcal{N} \) also \( z_y \) will be in \( \mathcal{N} \).

Not requiring \( \mathcal{N} \) to be downwards closed? (So \( \mathcal{N} = \mathcal{EB} \).) 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 \( e = \text{let ch=channel()} \) 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
\[
[\beta \sqsupseteq x; \text{CHAN}, \forall \sub{B}; x > \text{bool}, \forall \sub{B}; x > \text{int}]
\]
which are unsolvable since for a solution substitution \( \psi \) it will hold (with \( B = \text{fv}(\beta[\psi]) \)) that \( \forall B. x[\psi] > \text{bool} \) and \( \forall B. x[\psi] > \text{int} \); in addition we have \( \text{fv}(z[\psi]) \subseteq B \) so it even holds that \( z[\psi] \text{ bool and } z[\psi] \text{ int.} \). On the other hand, the algorithm from Figure 4 and 5 applied to \( e \) will fail immediately (since \( x \) 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 \( \mathcal{C} \) which is principal in the sense that for any \( \psi \) satisfying \( C \) it holds that \( \psi = \mathcal{C} \); \( \psi \) (so defining \( \mathcal{N} \) as \( \text{fv}(\mathcal{EB}[\mathcal{C}]) \) will establish (3) and (4)). Essentially \( \mathcal{C} \) maps \( \beta \) to \( B \cup \{ \beta \} \) if \( \beta \sqsupseteq B \) is in \( C \); our \( \mathcal{EB}[\mathcal{C}] \) therefore corresponds to the \( \text{fv}(\mathcal{EB}[\mathcal{C}]) \) 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 \( \forall T. \vec{x} \beta' > \vec{t} \beta' \) where \( \vec{x} \) 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 \( e; e; \ldots \), 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), \( \equiv \) stands for \( \equiv \), \( e \) stands for \( e \), 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 \( \psi \) that satisfies \( C \) is to proceed sequentially:

- if \( C \) is empty, let \( \psi = \text{id} \);
- otherwise, let \( C \) be the disjoint union of \( C' \) and \( C'' \). Suppose \( \psi' \) satisfies \( C' \) and suppose \( \psi'' \) 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 \( \phi \). This will be the case if \( C'[\psi'] \) only contains C-constraints (due to Fact 3.2) and S-constraints of form \( \forall F. \vec{g} \succ \vec{g} \) (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:

\[
\text{only solve S-constraints when they turn into S-equalities.} \quad (14)
\]

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

\[
\beta \equiv !z, \ \beta' \equiv !z, \ \forall \vec{0}.(x, \beta) \succ (\text{int}, \beta'), \ \forall \vec{0}.(x, \beta) \succ (\text{bool}, \beta'). \quad (15)
\]

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 \( \psi \) which maps \( \beta \) as well as \( \beta' \) into \( !z \). One might thus be tempted to think that \( \psi \) is a solution to the constraints in (15); but by applying \( \psi \) to these constraints we get

\[
\forall \vec{0}.(x, !z) \succ (\text{int}, !z), \ \forall \vec{0}.(x, !z) \succ (\text{bool}, !z)
\]

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 \( \forall F. (\vec{x}, \vec{\beta}) \succ (\vec{x}', \vec{\beta}') \), 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 \( (\vec{x}, \vec{x}') \) and \( (\vec{\beta}, \vec{\beta}') \)), 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 \( \beta \equiv b \) (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 \(\{\beta \supseteq b_1, \ldots, \beta \supseteq b_n\}\) 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 \(\beta\) into \(\text{REC}\beta. (b_1 + \ldots + b_n)\) (if \(\beta\) 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 \(b2 -> \text{rec } b2.(e+(r2\_chan\_a14\_list;fork_((b2;r2!a14\_list));b17;r2?a14\_list))\)

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, for instance, 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.

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 \( E \vdash e : t \& b \). Then also \( \kappa(E) \vdash e : t \& b \).

Proof
Induction in the proof tree. The only interesting case is ‘let’:

Suppose that \( E \vdash \text{let } x = e_1 \text{ in } e_0 : t \& b \) because \( E \vdash e_1 : t_1 \& b_1 \) and because \( E \oplus [x : \forall F.t_1] \vdash e_0 : t \& b_0 \) and because \( b \equiv b_1 ; b_0 \), where \( F = \text{fv}(E) \cup \text{fv}(b_1) \).

By induction it holds that

\( \kappa(E) \vdash e_1 : t_1 \& b_1 \)
and that $\kappa(E) \oplus [x : \forall F.t_1] \vdash e_0 : t \& b_0$; where $F' = F \cap \text{fv}(t_1)$.

Let $F'' = (\text{fv}(\kappa(E)) \cup \text{fv}(b_1)) \cap \text{fv}(t_1)$; then $F'' \subseteq F'$. Fact 4.9 then tells us that

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

which is enough to show the desired judgement

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

**Lemma B.2**

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

Before embarking on the proof, we first need an auxiliary concept: with $ts = \forall F.t$ and $ts' = \forall F'.t'$ type schemes, we say that $ts \equiv ts'$ (to be read ‘$ts$ is alpha-equivalent to $ts'$’) if and only if $ts' = ts[y]$ where $\psi$ is a total bijective mapping from variables into variables such that $F \cap \text{dom}(\psi) = 0$ (so $F' = F$ and $F \cap \text{ran}(\psi) = 0$ and $t' = t[y]$). Clearly this is an equivalence relation. We say that $E \equiv E'$ holds if and only if $\text{dom}(E) = \text{dom}(E')$ and for all $x \in \text{dom}(E)$ we have $E(x) \equiv E'(x)$ (so if $E \equiv E'$ then $\text{fv}(E) = \text{fv}(E')$). Some auxiliary results:

**Fact B.3**

Let $\psi$ be a total bijective mapping from variables into variables. Then

(a) If $ts > t'$ then also $ts[y] > t'[y]$;

(b) if $ts > t'$ with $ts$ a closed extended type scheme then also $ts > t'[y]$;

(c) $\text{gen}(t_1, E, b_1)[y] = \text{gen}(t_1[y], E[y], b_1[y])$.

**Proof**

Let $\psi^{-1}$ be the inverse of $\psi$. For (a), write $ts = \forall F.t$ such that $ts[y] = \forall F'.t[y]$ with $F' = \text{fv}(F[y])$. There exists an instance substitution $\phi$ with $\text{dom}(\phi) \cap F = 0$ such that $t' = t[\phi]$. Now define $\phi' = \psi^{-1} \circ \phi \circ \psi$; this $\phi'$ will suffice as instance substitution since

- $t[y]/[\phi'] = t[\phi][\psi] = t'[y]$;
- for $\gamma \in F'$ we have (as $\gamma[\psi^{-1}] \in F$) that $\gamma[\phi'] = \gamma[\psi^{-1} \circ \phi \circ \psi] = \gamma$, showing that $\text{dom}(\phi') \cap F' = 0$.

For (b), write $ts = \forall (t, C)$. There exists an instance substitution $\phi$ such that $t' = t[\phi]$ and such that $\phi \models C$. This shows that we have $ts > t'[y]$ 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 $F = \text{fv}(t_1) \cap (\text{fv}(E) \cup \text{fv}(b_1))$. The result then follows from the calculation

$$\gamma \in \text{fv}(F[y])$$

$$\iff \exists \gamma' \text{ such that } \gamma = \gamma'[y] \text{ and } \gamma' \in F$$

$$\iff \gamma[\psi^{-1}] \in F$$

$$\iff \gamma[\psi^{-1}] \in \text{fv}(t_1) \text{ and } (\gamma[\psi^{-1}] \in \text{fv}(E) \text{ or } \gamma[\psi^{-1}] \in \text{fv}(b_1))$$

$$\iff \gamma \in \text{fv}(t_1[y]) \text{ and } (\gamma \in \text{fv}(E[y]) \text{ or } \gamma \in \text{fv}(b_1[y]))$$

$$\iff \gamma \in \text{fv}(t_1[y]) \cap (\text{fv}(E[y]) \cup \text{fv}(b_1[y]))$$
Fact B.4
Suppose \( E \vdash e : t \& b \); and suppose that \( \psi \) is a total bijective mapping from variables into variables. Then also \( E[\psi] \vdash e : t[\psi] \& b[\psi]. \)

Proof
A straightforward induction in the proof tree, using Fact B.3 and Fact 3.2.

Fact B.5
Suppose that \( E \models E' \) 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 \( E(x) > t \) and because \( b \supseteq e \). Let \( E(x) = \forall t x \); there thus exists \( \phi \) with \( \operatorname{dom}(\phi) \cap F = \emptyset \) such that \( t = t_x[\phi] \). We have \( E'(x) = \forall t'_x \), where \( t'_x = t_x[\psi] \) with \( \psi \) a total bijective mapping from variables into variables such that \( \operatorname{dom}(\psi) \cap F = \emptyset \).

Now define \( \psi' \) as follows: if \( \gamma \in \operatorname{fv}(t'_x) \) then \( \gamma[\psi'] = \gamma[\psi^{-1}; \phi] \); and \( \gamma[\psi'] = \gamma \) otherwise. It is clear that \( t'_x[\psi'] = t_x[\phi] = t \) and that \( \operatorname{dom}(\psi') \cap F = \emptyset \); 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 \( E(x) > t \) and because \( b \supseteq e \). Let \( E(x) = \forall t x \); then there exists a \( \phi \) with \( \operatorname{dom}(\phi) \cap F = \emptyset \) such that \( t = t_x[\phi] \). We have \( E'(x) = \forall t'_x \), with \( F' \cap \operatorname{fv}(t_x) = F \). Now let \( \phi' \) behave as \( \phi \) on \( \operatorname{fv}(t_x) \) and as the identity otherwise; then \( \operatorname{dom}(\phi') \cap F' = \emptyset \) and \( t_x[\phi'] = t \). This shows that \( E'(x) > t \) and hence \( E' \vdash x : t \& b \).

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

Let \( \bar{\gamma} = \operatorname{fv}(t_1) \setminus (\operatorname{fv}(E) \cup \operatorname{fv}(b_1)) \); and let \( \bar{\gamma}' \) be ‘fresh’ copies of \( \bar{\gamma} \). Let \( \psi \) be a substitution which maps \( \bar{\gamma} \) into \( \bar{\gamma}' \); which maps \( \bar{\gamma}' \) into \( \bar{\gamma} \) and which otherwise behaves as the identity. By Fact B.4 it holds (since \( \operatorname{dom}(\psi) \cap \operatorname{fv}(b_1) = \emptyset \)) that \( E[\psi] \vdash e_1 : t_1[\psi] \& b_1 \).

It is easy to see (since \( \operatorname{dom}(\psi) \cap \operatorname{fv}(E) = \emptyset \)) that \( E[\psi] \models E \) 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

\[
E' \vdash_2 e_1 : t_1[\psi] \& b_1.
\]

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

\[
\kappa(E' \oplus [x : \forall F.t_1[\psi]]) = E \oplus [x : \forall F.t_1[\psi]].
\]

Since \( \operatorname{dom}(\psi) \cap F = \emptyset \) we conclude that \( \forall F.t_1 \models \forall F.t_1[\psi] \). Fact B.5 now tells us that

\[
E \oplus [x : \forall F.t_1[\psi]] \vdash e_0 : t \& b_0.
\]
Due to (17) we can apply the induction hypothesis on (18) to get
\[ E' \odot [x : \forall T. t_1[y]] \vdash e_0 : t \land b_0 \]
and by combining (16) and (19) we arrive at the desired judgement
\[ E' \vdash_2 \text{let } x = e_1 \text{ in } e_0 : t \land b. \]

References


