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

November 13, 1995

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.

Chapter 1

Introduction

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

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

In [NN93] and [NN94a] inference systems are developed that extend the usual notion of types with behaviours. Applications of such information are

demonstrated in [NN94a] and [NN94c].

The question remains: how to implement the inference system, i.e. how to reconstruct the types and behaviours? It seems suitable to use a modified version of algorithm W [Mil78]. This algorithm works by unification, but since our behaviours constitute a non-free algebra (due to the laws imposed on them) this approach is not immediately feasible in our framework. Instead we employ the technique of algebraic reconstruction [JG91, TJ92]; that is the algorithm unifies the free part of the type structure and generates constraints to cater for the non-free parts.

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

The present paper improves on [NN94b] by (i) achieving completeness in addition to soundness, by means of another generalisation strategy (made possible by a different formulation of S-constraints); (ii) avoiding some redundancy in the generated constraints (and in the correctness proofs). 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". This is related to the fact that S-constraints can be viewed as a special case of semi-unification.

Overview of paper

Chapter 2 and 3 set up the background for the present work: in Chapter 2 we give a brief motivating introduction to CML and behaviours, and in Chapter 3 we present the inference system from [NN94a]. Chapter 4 contains a detailed motivation for our design of the reconstruction algorithm W. Chapter

5 elaborates on our choice of generalisation strategy. In Chapter 6 and 7 the algorithm is shown to be sound and complete; the proofs can be found in Appendix B and C. In Chapter 8 we show how to solve the constraints generated for some special cases. Chapter 9 concludes. Example output from our prototype implementation is shown in Appendix A.

Chapter 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 this channel. The purpose of the constant sync is to convert a communication *possibility* into an actual communication (see [Rep91] 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 $(\alpha_1 \to \alpha_2) \to (\alpha_1\ \mathbf{list} \to \alpha_2\ \mathbf{list})$ but we can annotate this type with *behaviour* information yielding 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:

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

So either b_2 performs no communication (if the list xs is empty) or it will first allocate a channel which transmits values of type α_2 **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 α_2 **list**; then it performs β_1 (by computing f on the head of the list); and finally it receives a value of type α_2 **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 is a consequence of the subject reduction theorem from [NN94a].

Some useful analyses on behaviours. In [NN94a] a behaviour is tested for 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 lead to substantial savings in the run-time system. In [NN94c] 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 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 communications take place); an output action $!t$ (a value of type t is sent); an input action $?t$ (a value of type t is received); a channel action t CHAN (a channel able to transmit values of type t is created); a fork action 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 automatically.

Compared to [NN94a] we have omitted *regions* as these present no additional problems to the algorithm.

Chapter 3

The type and behaviour inference system

In Fig. 3.1 (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 is always the case for program analysis we shall be interesting 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. [Tan94]) but not "subtyping". To formalise this we impose a preorder $\sqsupseteq$ on behaviours just as in [NN94a, 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.) This approximation is "inlined" in 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 Fig. 3.2, where $b_1 \equiv b_2$ denotes that $b_1 \sqsupseteq b_2$ and $b_2 \sqsupseteq b_1$ 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).

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

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

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

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

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

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

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

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

Figure 3.1: The type and behaviour inference system.

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	$\mathsf{REC}\beta.b \equiv b[\beta \mapsto \mathsf{REC}\beta.b]$		

Figure 3.2: The preorder $\sqsupseteq$ with equivalence $\equiv$.

Fact 3.2 If $b_1 \sqsupseteq b_2$ then $\mathsf{fv}(b_1) \supseteq \mathsf{fv}(b_2)$ and $b_1[\phi] \sqsupseteq b_2[\phi]$.

We now return to Fig. 3.1. The clause for monomorphic abstractions 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 applications 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). The clause for polymorphic abstractions $\mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0$ 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 $\mathsf{gen}(t_1, E, b_1)$ which creates a type scheme whose type part is t_1 and where all variables are polymorphic 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 [TJ94]). 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 a closed *extended type scheme*, that is in addition to a type it also contains a set of constraints of form $b_1 \sqsupseteq b_2$. Such constraints are denoted *C-constraints* (for containment). The value of CTypeOf() on some constants (all occurring in Example 2.1) is tabulated below (adopted from [NN94b,

Table 4]).

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

The inference system is much as in [NN94a] (whereas the inference system in [NN93] uses subtyping instead of polymorphism), but in [NN94a] the C-constraints present in the definition of CTypeOf() have been "coded into" the types.

Chapter 4

Designing the reconstruction algorithm W

Our goal is to produce an algorithm which works in the spirit of the well-known algorithm W [Mil78], 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 introduce the notion of S-constraints which is needed since C-constraints in general have no principal solution; in Section 4.3 we improve on the algorithm from [NN94b] 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 (and correctness proof) 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 3.2 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 [TJ92, NN94b] 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 iff t_1 and t_2 are both simple and $b = \beta$ for some β); a behaviour is simple if all the types it contains are simple (so e.g. $!t$ is simple iff t is simple) and if it does not contain sub-behaviours of form $\mathsf{REC}\beta.b$; a C-constraint is simple if it is of form $\beta \sqsupseteq b$ 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).

Fact 4.1 Simple types are closed under the application of simple substitution: $t[\phi]$ 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 ϕ and ϕ' are. □

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

We can now define a procedure UNIFY which takes two simple types t_1 and t_2 and returns the most general unifier if one unifier exists – otherwise UNIFY fails. 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 are variables due to the types being simple).

So the precise definition of UNIFY will contain the cases below:

$$\mathsf{UNIFY}(\alpha, t) = [\alpha \mapsto t] \text{ provided } t = \alpha \text{ or } \alpha \notin \mathsf{fv}(t)$$

$$\begin{aligned} \mathsf{UNIFY}(t_1 \ \mathbf{com}\ \beta_1, t_2 \ \mathbf{com}\ \beta_2) &= \theta'; [\beta_1' \mapsto \beta_2'] \text{ where} \\ \mathsf{UNIFY}(t_1, t_2) &= \theta' \text{ and } \beta_i' = \beta_i[\theta'] \end{aligned}$$

Fact 4.3 If UNIFY is called on simple types, all arguments to subcalls will be simple types.

The substitution returned by UNIFY is simple. □

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

Lemma 4.4 Suppose $\mathsf{UNIFY}(t_1, t_2) = \theta$. 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 $t = \alpha$).

Now 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 $t_1[\theta'] = t_2[\theta']$ and hence

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

Lemma 4.5 Suppose $t_1[\psi] = t_2[\psi]$ (with t_1,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 size 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 $\psi = [\alpha \mapsto t]; \psi$.

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 $\beta_i' = \beta_i[\theta']$ 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 The need for S-constraints

The most distinguishing feature of our approach is the presence of the so-called *S-constraints* (for substitutions), whose purpose is to record that the solution chosen for polymorphic variables and their instances must be in the same "solution class" with the solution for the polymorphic variables "principal" in that class. This is necessary because there seems to be no notion of "principal solutions" to C-constraints. For an example of 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 as 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.[1]

[1] 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 Fig. 3.2 which states that $\mathsf{REC}\beta.b$ is equivalent

As we shall see in Chapter 8 it is always possible to find *a* solution to a given set of C-constraints, but due to the lack of principality such a solution may not correspond to a valid inference: if β is a polymorphic variable occurring in the type of a let-bound identifier x and β' is a copy of β made when analysing an instance of x then (as the constraints for β are copied too) any solution to the constraints for β' is a copy of a solution to the constraints for β (and hence an instance of the principal solution if a such existed), but the solution actually *chosen* for β needs not have the solution chosen for β' as instance. Hence we need to record, by means of an S-constraint, that the solution chosen for β should have the solution chosen for β' as an instance.

The above considerations motivated the design of the algorithm in [NN94b] where the environment binds each identifier x to a type scheme[2] of form $\forall\vec{\gamma}.(t, C)$ and where the following actions are taken when such an x is met: copies t' and C' of t and C are created, where $\vec{\gamma}$ has been replaced by fresh variables $\vec{\gamma}'$; and an S-constraint $\vec{\gamma} \succ \vec{\gamma}'$ is generated.

An additional feature present in [NN94b], 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 must not "mix" those variables. This requirement has severe impact on which variables to quantify (i.e. make polymorphic) 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 (these variables will be called $\mathcal{EB}$ in the rest of the paper) are not quantified over one will also need to ensure that the set of variables not quantified over (these variables will be denoted $\mathcal{NQ}$ in the rest of the paper) is *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 (simple) C-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 (simple) C-constraints. We say that F is upwards closed wrt. C if the following property

to its finite unfoldings, and cf. [CC91] where a similar change in axiomatisation is made concerning recursive types).

[2]We use γ to range over type variables and behaviour variables collectively and use g to range over types and behaviours collectively.

holds for all $\beta \sqsupseteq b \in C$: if $\mathsf{fv}(b) \cap F \neq \emptyset$ then $\beta \in F$. $\square$

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. Similarly for the "downwards and upwards closure".

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 less reasonable).

4.3 Achieving completeness

The last remarks in the preceding section suggest that the proper demand to $\mathcal{NQ}$ 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. Hence this restriction has to be weakened; and it turns out that a suitable "degree of distinction" between the two kinds of variables can be coded into the S-constraints by letting them take the form $\forall \overline{F}.\vec{g} \succ \vec{g}'$ (with F a set of variables which one should think of as non-polymorphic). Such a constraint is satisfied by a substitution ψ iff $\vec{g}'[\psi]$ is an instance of $\vec{g}[\psi]$, i.e. of form $\vec{g}[\psi][\phi]$, where the domain $\mathsf{dom}(\phi)$ of the instance substitution ϕ is disjoint from $\mathsf{fv}(F[\psi])$. This explains S-constraints as a special case of semi-unification.

4.4 Eliminating redundancy

S-constraints are introduced when meeting an identifier x which is bound to a type scheme $\forall \vec{\gamma}.(t, C)$; then the 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 the

copying of 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 (as 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

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 (so $\mathsf{fv}(ts) = F$). 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* and corresponds to the inference system; type schemes which are not necessarily kernel are said to be *enriched* (and are essential for the algorithm).

The instance relation is defined in a way consistent with the classical definition: $\forall\overline{F}.t \succ t'$ holds iff there exists a substitution ϕ with $\mathsf{dom}(\phi) \cap F = \emptyset$ such that $t' = t[\phi]$. For extended type schemes we say that $\forall \ldots (t, C) \succ t'$ holds iff there exists ϕ such that ϕ satisfies C (as usual this is written $\phi \models C$) and $t' = t[\phi]$.

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 $t = t'$ and $F \subseteq F'$. As expected we have

Fact 4.8 *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 these newly defined entities:[3] if $ts = \forall\overline{F}.t$ then $ts[\psi] = \forall\overline{F'}.t[\psi]$ where $F' = \mathsf{fv}(F[\psi])$; and the re-

[3] As long as substitutions do not affect the bound variables (which we can prove will

sult 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 iff the type $t'[\psi]$ is an instance of the (enriched) type scheme $(\forall\overline{F}.t)[\psi]$.

4.6 Algorithm W

We are now ready to define the reconstruction algorithm W, as is done in Figure 4.1 and 4.2. The algorithm fails if and only if a call to UNIFY fails. W takes as input a CML-expression and an environment where all types are simple and returns as output a simple type, a simple behaviour, a list of constraints where the C-constraints are simple, and a simple substitution.

Most parts of the algorithm are either standard or have been explained earlier in this chapter. 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*.

be the case for the substitutions produced by our algorithm) the usual laws still hold, e.g. that if $ts \succ t$ then $ts[\psi] \succ t[\psi]$ but this result is not needed for our correctness proofs.

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

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

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

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

Figure 4.1: Algorithm W, first part.

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

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

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

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

Figure 4.2: Algorithm W, second part.

Chapter 5

Choice of generalisation strategy

It turns out that in order to prove soundness (as is done in Appendix B) all we need to know about $\mathcal{NQ}$ is that

$$\psi \models C_1 \Rightarrow \mathsf{fv}(\mathcal{NQ}[\psi]) \supseteq \mathsf{fv}(\mathcal{EB}[\psi]) \tag{5.1}$$

and it turns out that in order to prove completeness (as is done in Appendix C) all we need to know about $\mathcal{NQ}$ is that

$$\psi \models C_1 \Rightarrow \mathsf{fv}(\mathcal{NQ}[\psi]) \subseteq \mathsf{fv}(\mathcal{EB}[\psi]) \ . \tag{5.2}$$

From Fact 3.2 it is immediate to verify that these two properties indeed hold for $\mathcal{NQ}$ as defined in Figure 4.2. In this chapter we shall investigate whether other definitions of $\mathcal{NQ}$ might be appropriate.

Requiring $\mathcal{NQ}$ to be upwards closed. (As already mentioned this is essentially what is done in [NN94b].) Then (5.1) will still hold so soundness is assured. On the other hand (5.2) 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 we have $\mathcal{NQ} = \mathcal{EB}$.) It is trivial that (5.1) and (5.2) 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 $e =$ 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

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

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 Figure 4.1 and 4.2 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.

Note that e above can be typed if ch is assigned the type scheme $\forall\overline{\emptyset}.\alpha$ **chan** but cannot be typed if ch is assigned the type scheme $\forall\overline{\{\alpha\}}.\alpha$ **chan**. The former case will arise if α **chan** is handled as α **list**; whereas the latter case will arise if α **chan** is handled by the techniques for α **ref** developed in [TJ94] and [BD93].

The approach in [TJ94].

We have seen that there are several possibilities for satisfying (5.1) and (5.2); so settling on the downwards closure as we have done may seem somewhat arbitrary but can be justified by observing the similarities to what is done in [TJ94].

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 $\psi = \overline{C}; \psi$. What corresponds to our $\mathcal{NQ}$ is then essentially defined as $\mathsf{fv}(\mathcal{EB}[\overline{C}])$ and notice that since $\overline{C}$ is principal as defined above (5.1) and (5.2) hold.

The principal solution $\overline{C}$ basically for each $\beta \supseteq B \in C$ maps β into $B \cup \{\beta\}$; so applying $\overline{C}$ corresponds to taking the downwards closure.

Chapter 6

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 3.1.

Theorem 6.1 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 $\emptyset \vdash e : t' \;\&\; b'$. □

This theorem follows easily (using Fact 3.1) from Proposition 6.2 below that allows an inductive proof, to be found in Appendix B. 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 6.2 Suppose that $W(e, E) = (t, b, C, \theta)$. Then for all ψ with $\psi \models C$ we have $\kappa(E[\theta][\psi]) \vdash e : t[\psi] \;\&\; b[\psi]$. □

Chapter 7

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 6.1:

Theorem 7.1 Suppose $\emptyset \vdash e : t' \ \& \ b'$.
Then $W(e, \emptyset)$ succeeds with result (t, b, C, θ)
and $\exists\ \psi$ 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. 3.1 (ii) prove the algorithm complete wrt. this inference system.

The problem with the original system is that generalisation is defined as $\mathsf{gen}(t_1, E, b_1) = \forall \overline{F}.t_1$ where $F = (\mathsf{fv}(E) \cup \mathsf{fv}(b_1)) \cap \mathsf{fv}(t_1)$; 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 old one except that $F = \mathsf{fv}(E) \cup \mathsf{fv}(b_1)$. Hence inferences will be of form $E \vdash_2 e : t \ \& \ b$ where the environment E may now contain *enriched* type schemes. We have the desired equivalence result, to be proved in Appendix D:

Proposition 7.2 Assume $\kappa(E') = E$. Then $E \vdash e : t \ \& \ b$ holds iff it holds that $E' \vdash_2 e : t \ \& \ b$. (In particular, $\emptyset \vdash e : t \ \& \ b$ iff $\emptyset \vdash_2 e : t \ \& \ b$.) □

So in order to prove Theorem 7.1 it will be sufficient to show Proposition 7.3 below that allows an inductive proof, to be found in Appendix C. Often (as

in e.g. [Jon92]) the assumptions in a completeness proposition are (using the terminology of Prop. 7.3) that $E[\phi]$ is *equal to* E'; but as in [Smi93] this is not sufficient since 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 so one may produce behaviours containing many "extra" variables which cannot be quantified over in let-expressions; whereas in [Smi93] it is due to the fact that the inference system gives freedom to quantify over fewer variables than possible.)

Proposition 7.3 Suppose $E' \vdash_2 e : t' \,\&\, b'$ and $E[\phi] \succeq E'$. Then $W(e, E)$ succeeds with result (t, b, C, θ) and there exists a ψ such that[1] $\theta; \psi \stackrel{E}{=} \phi$ and $\psi \models C$ and $t' = t[\psi]$ and $b' \sqsupseteq b[\psi]$. □

[1]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 $\mathsf{var}(E)$ are *all* the variables (i.e. $\mathsf{var}(\forall \overline{F}.t) = \mathsf{fv}(t) \cup F$).

Chapter 8

Solving the constraints

In this chapter we discuss how to solve the constraints generated by Algorithm W. We have seen that the C-constraints are simple and hence of form $\beta \sqsupseteq b$ with b a simple behaviour; and at some effort one can show 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 Fig. 3.2 so as to decrease (in most cases) 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; $>$ stands for $\sqsupseteq$; e stands for ϵ; the r_i are "region variables" and can be omitted).

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

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

solving" to be correct:

S-constraints are solved only when they become S-equalities. (8.1)

To see why sequential solving may go wrong if (8.1) is not imposed consider the constraints below:

$$\beta \sqsupseteq !\alpha,\ \beta' \sqsupseteq !\alpha,\ \forall\overline{\emptyset}.(\alpha,\beta)\ \succ\ (\mathsf{int},\beta'),\ \forall\overline{\emptyset}.(\alpha,\beta)\ \succ\ (\mathsf{bool},\beta'). \tag{8.2}$$

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 (8.2); 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 section (we shall not deal with other kinds of S-constraints even though some of those might have simple solutions as well).

First we convert the S-constraints into S-equalities, cf. (8.1). This is done by identifying all type and behaviour variables occurring in "corresponding positions" (thus going from a polymorphic world into a monomorphic world).

Next we have to solve the C-constraints sequentially; and 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 [NN94b]: due to rule R1 in Figure 3.2 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 (after appropriate line-breaking):

```
*** 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) was what we expected.

Solving constraints in the general case. One can code up solving S-constraints as a semi-unification problem and though the latter problem is undecidable several decidable subclasses exist. We expect our S-constraints to fall into one of these (since they are generated in a "structured way") and hence one might be tempted to use e.g. the algorithm for semi-unification described in [Hen93]. But this is not enough since we in addition have to solve the C-constraints, and as witnessed by the constraints in (8.2) this may destroy the solution to the S-constraints.

Chapter 9

Conclusion

In this paper we have adapted the traditional algorithm W to apply to our type and behaviour system. We have improved upon a previously published algorithm [NN94b] in achieving completeness and eliminating some redundancy in representation. The algorithm has been implemented and has provided quite illuminating analyses of example CML programs.

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

Another and major difference from the traditional formulation, as well as that of [JG91], is that we generate so-called S-constraints that also have to be solved. This phenomenon is needed because our C-constraints would seem not to have principal solutions. This is not the case for the traditional "free" unification of Standard ML, but it is a phenomenon well-known in unification theory [Sie89]. As a consequence we have to ensure that the different solutions to the C-constraints (concerning the polymorphic definition and its instantiations) are comparable and this is the purpose of the S-constraints. Solving S-constraints is a special case of semi-unification and even though the latter is undecidable we may expect 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.

The approach pursued is this paper is to accept that there are constraints

which do not have principal solutions; future work along this avenue is to develop heuristics and/or algorithms for solving the resulting S-constraints.

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

Bibliography

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

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

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

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

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

[Jon92] Mark P. Jones. A theory of qualified types. In *ESOP '92*, volume 582 of *LNCS*, pages 287–306. Springer-Verlag, 1992.

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

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

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

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

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

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

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

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

[Smi93] Geoffrey S. Smith. Polymorphic type inference with overloading and subtyping. In *TAPSOFT '93*, volume 668 of *LNCS*, pages 671–685. Springer-Verlag, 1993.

[Tan94] 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.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[TJ94] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2), 1994.

Appendix 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
```

Appendix B

Proof of Proposition 6.2.

PROOF: We perform induction on e; to conserve space we use the same terminology as in the definition of the relevant clause for W. 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

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

Since $\psi \models C$ we have $\forall \overline{F'}.\vec{\gamma}[\psi] \succ \vec{\gamma}'[\psi]$; so there exists a ϕ' with $\mathsf{dom}(\phi') \cap F' = \emptyset$ such that $\psi;\phi'$ 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 (B.1) 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 $\forall \ldots (t_c, C_c) \succ t_c[[\vec{\gamma} \mapsto \vec{\gamma}'];\psi]$. 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 judgments enable us to arrive at the desired judgment

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

The case $W(\lambda x.e_0, E)$**:** We can apply the induction hypothesis to get

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

and since $\beta_0[\psi] \sqsupseteq b_0[\psi]$ we due to Fact 3.1 also 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

$$\kappa(E[\theta][\psi]) \ \vdash \ e_1 : t_1[\theta_0; \psi] \ \& \ b_1[\theta_0; \psi]. \tag{B.2}$$

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

$$\kappa(E[\theta][\psi]) \oplus [x : \forall \overline{F'}.(t_1[\theta_0; \psi])] \ \vdash \ e_0 : t_0[\psi] \ \& \ b_0[\psi] \tag{B.3}$$

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

Let $F'' = (\mathsf{fv}(\kappa(E[\theta][\psi])) \cup \mathsf{fv}(b_1[\theta_0;\psi])) \cap \mathsf{fv}(t_1[\theta_0;\psi])$. As (5.1) holds for our choice of $\mathcal{NQ}$, we can use it on the substitution $\theta_0;\psi$ to infer that $F'' \subseteq F'$. Hence we can apply Fact 4.8 on (B.3) to get

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

which together with (B.2) is enough to show the desired judgment

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

□ □

Appendix C

Proof of Proposition 7.3.

PROOF: 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. 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

$$\mathsf{dom}(\phi') \cap \mathsf{fv}(F_x[\phi]) = \emptyset. \tag{C.1}$$

Now define ψ as follows: it maps $\vec{\gamma}'$ into $\vec{\gamma}[\phi;\phi']$; and otherwise it behaves like ϕ. This ensures that $\theta;\psi \overset{E}{=} \phi$; and it is trivial that $b' \sqsupseteq b[\psi]$. For our remaining claims, observe that due to (C.1) we have

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

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

The case $e = c$**:** Let $\mathrm{CTypeOf}(c) = \forall \ldots (t_c, C_c)$. 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 $e = e_1\ e_2$**:** Some terminology: we define $V_1 = \mathsf{var}(t_1, b_1, C_1, \theta_1)$ and $V_2 = \mathsf{var}(t_2, b_2, C_2, \theta_2)$ and $E_1 = \mathsf{var}(E[\theta_1])$; and say that a variable is "internal" if it occurs in V_1 but not in E_1. We can assume that no internal variable occurs in V_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 find 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 $E[\theta_1][\psi_1] \succeq E'$ so we can apply the induction hypothesis once more to find ψ_2 such that $\theta_2; \psi_2 \stackrel{E_1}{=} \psi_1$; such that $\psi_2 \models C_2$; such that $t'_2 = t_2[\psi_2]$ and such that $b'_2 \sqsupseteq b_2[\psi_2]$.

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

We have the following relations:

$$\psi_0 \stackrel{V_2}{=} \psi_2 \text{ and } \theta_2; \psi_0 \stackrel{V_1 \cup E_1}{=} \psi_1 \tag{C.2}$$

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 (C.2) 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 and that there exists ψ such that $\psi_0 = \theta_0; \psi$. Using this and (C.2) 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 Fig. 3.2) 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 $E' \oplus [x : t'_1] \vdash_2 e_0 : t'_0 \;\&\; b'_0$ and because $t' = t'_1 \to^{b'_0} t'_0$ and because $b' \sqsupseteq \epsilon$. 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 find 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

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

The case $e = \mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0$**:** First some terminology: we define $V_1 = \mathsf{var}(t_1, b_1, C_1, \theta_1)$ and $V_0 = \mathsf{var}(t_0, b_0, C_0, \theta_0)$ and $V_e = \mathsf{var}(E[\theta_1]) \cup \mathcal{NQ} \cup \mathsf{fv}(t_1)$; and say that a variable is "internal" if it occurs in V_1 but not in V_e. We can assume that no internal variable occurs in V_0.

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

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

for since (5.2) holds (cf. Chapter 5) we have $\mathsf{fv}(\mathcal{NQ}[\psi_1]) \subseteq \mathsf{fv}(\mathsf{fv}(E[\theta_1])[\psi_1]) \cup \mathsf{fv}(b_1[\psi_1])$, by assumption $\mathsf{fv}(\mathsf{fv}(E[\theta_1])[\psi_1]) = \mathsf{fv}(E[\theta_1][\psi_1]) = \mathsf{fv}(E[\phi]) \subseteq \mathsf{fv}(E')$, and from Fact 3.2 we have $\mathsf{fv}(b_1[\psi_1]) \subseteq \mathsf{fv}(b'_1)$. (Here we see the need for $\succeq$, cf. the discussion in Chapter 7.)

Using the above observation it is easy to verify that

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

so by induction we see that $W(e_0, \_)$ succeeds and that there exists ψ_0 such that $\theta_0; \psi_0$ equals ψ_1 on V_e, such that $\psi_0 \models C_0$, such that $t' = t_0[\psi_0]$ and such that $b'_0 \sqsupseteq b_0[\psi_0]$.

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

$$\psi \text{ equals } \psi_0 \text{ on } V_0 \text{ and } \theta_0; \psi \text{ equals } \psi_1 \text{ on } V_1 \cup V_e \tag{C.3}$$

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 (C.3) 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 $b' \sqsupseteq b'_1; b'_0 \sqsupseteq b_1[\psi_1]; b_0[\psi_0] = b_1[\theta_0][\psi]; b_0[\psi] = b[\psi]$. □ □

Appendix D

Proof of Proposition 7.2.

The proposition follows from the two lemmas below:

Lemma D.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 4.8 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 judgment

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

□

Lemma D.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 $s = \forall\overline{F}.t$ and $s' = \forall\overline{F'}.t'$ type schemes, we say that $s \stackrel{\alpha}{=} s'$, to be read "$s$ is alpha-equivalent to s'", holds iff $F = F'$ and $t' = t[\psi]$ where ψ is a bijective mapping from variables into variables such that $F \notin \mathsf{var}(\psi)$. Two auxiliary results:

Fact D.3 Suppose $E \vdash e : t \;\&\; b$; and suppose that ψ is a 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. □

Fact D.4 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 being 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 bijective mapping from variables into variables such that $\mathsf{var}(\psi) \cap F = \emptyset$.

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 D.2:

PROOF: Structural induction in e. 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' be the restriction of ϕ to $\mathsf{fv}(t_x)$; 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 $E \vdash \mathsf{let}\ x{=}e_1\ \mathbf{in}\ e_0 : t \;\&\; b$ holds 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 $F = (\mathsf{fv}(E) \cup \mathsf{fv}(b_1)) \cap \mathsf{fv}(t_1)$.

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 D.3 it holds (exploiting $\mathsf{dom}(\psi) \cap \mathsf{fv}(b_1) = \emptyset$) that $E[\psi] \vdash e_1 : t_1[\psi] \;\&\; b_1$. It is easy to see (since $\mathsf{var}(\psi) \cap \mathsf{fv}(E) = \emptyset$) that $E[\psi] \stackrel{\alpha}{=} E$ and hence we by Fact D.4 conclude that $E \vdash e_1 : t_1[\psi] \;\&\; b_1$. The induction hypothesis now tells us that

$$E' \vdash_2 e_1 : t_1[\psi] \;\&\; b_1. \tag{D.1}$$

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

$$\kappa(E' \oplus [x : \forall \overline{F'}.t_1[\psi]]) = E \oplus [x : \forall \overline{F}.t_1[\psi]]. \tag{D.2}$$

Since $\mathsf{var}(\psi) \cap F = \emptyset$ we conclude that $\forall \overline{F}.t_1 \stackrel{\alpha}{=} \forall \overline{F}.t_1[\psi]$. Fact D.4 now tells us that

$$E \oplus [x : \forall \overline{F}.t_1[\psi]] \vdash e_0 : t \;\&\; b_0. \tag{D.3}$$

Due to (D.2) we can apply the induction hypothesis on (D.3) to get

$$E' \oplus [x : \forall \overline{F'}.t_1[\psi]] \vdash_2 e_0 : t \;\&\; b_0 \tag{D.4}$$

and by combining (D.1) and (D.4) we arrive at the desired judgment

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

□