Polymorphic Subtyping for Effect Analysis: the Integration

H.R.Nielson & F.Nielson & T.Amtoft

Computer Science Department, Aarhus University, Denmark

e-mail:{hrnielson,fnielson,tamtoft}@daimi.aau.dk

April 15, 1996

Abstract

The integration of polymorphism (in the style of the ML `let`-construct), subtyping, and effects (modelling assignment or communication) into one common type system has proved remarkably difficult. One line of research has succeeded in integrating polymorphism and subtyping; adding effects in a straightforward way results in a semantically unsound system. Another line of research has succeeded in integrating polymorphism, effects, and subeffecting; adding subtyping in a straightforward way invalidates the construction of the inference algorithm. This paper integrates all of polymorphism, effects, and subtyping into an annotated type and effect system for Concurrent ML and shows that the resulting system is a conservative extension of the ML type system.

1 Introduction

Motivation. The last decade has seen a number of papers addressing the difficult task of developing type systems for languages that admit polymorphism in the style of the ML `let`-construct, that admit subtyping, and that admit effects as may arise from assignment or communication.

This is a problem of practical importance. The programming language Standard ML has been joined by a number of other high-level languages demonstrating the power of polymorphism for large scale software development. Already Standard ML contains imperative effects in the form of `ref`-types that can be used for assignment; closely related languages like Concurrent ML or Facile further admit primitives for synchronous communication. Finally, the trend towards integrating aspects of object orientation into these languages necessitates a study of subtyping.

Apart from the need to type such languages we see a need for type systems integrating polymorphism, subtyping, and effects in order to be able to continue the present development of annotated type and effect systems for a number of static program analyses; example analyses include control flow analysis, binding time analysis and communication analysis. This will facilitate modular proofs of correctness while at the same time allowing the inference algorithms to generate syntax-free constraints that can be solved efficiently.

State of the art. One of the pioneering papers in the area is [8] that developed the first polymorphic type inference and algorithm for the applicative fragment of ML; a shorter presentation for the typed λ-calculus with `let` is given in [2].

Since then many papers have studied how to integrate subtyping. A number of early papers did so by mainly focusing on the typed λ-calculus and only briefly dealing with `let` [9, 4]. Later papers have treated polymorphism in full generality [15, 6]. A key ingredient in these approaches are the techniques for simplifying the enormous set of constraints into something manageable [3, 15].

Already ML necessitates an incorporation of imperative effects due to the presence of `ref`-types. A pioneering paper in the area is [18] that developes a distinction between imperative and applicative type variables and that characterises expressions as being expansive or non-expansive. A number of papers have tried to improve upon this work by allowing to type programs that are rejected according to the expansiveness distinction; this includes [7, 19, 16] but all of these systems (as well as the one we develop) fail to fully generalise the expansiveness distinction as is discussed in [16, section 11].

In the area of static program analysis, annotated type and effect systems have been used as the basis for variations of control flow analysis [17] and binding time analysis [11, 5]. These papers typically make use of a polymorphic type system with subtyping and no effects or a non-polymorphic type system with effects and subtyping. A more ambitious analysis is the approach of [12] to let annotated type and effect systems extract terms of a process algebra from programs with communication; this involves polymorphism and subeffecting but some algorithmic problems remain [10].

A step forward. In this paper we take an important step towards integrating polymorphism, subtyping, and effects into one common type system. As far as the annotated type and effect system is concerned this involves the following key idea:

- Carefully taking effects into account when deciding the set of variables over which to generalise in the rule for `let`; this involves taking upwards closure with respect to a constraint set and is essential for maintaining semantic soundness and a number of substitution properties.

This presents a major step forward in generalising the subeffecting approach of [16] and in admitting effects into the subtyping approaches of [15, 6]. The development is not only applicable to Concurrent ML (with communication) but also Standard ML (with references) and similar settings.

Overview. In this paper we study a fragment of Concurrent ML that includes the λ-calculus, `let`-polymorphism, and primitives for synchronous communication as well as the dynamic creation of channels and processes. We develop an annotated type and effect system in which a simple notion of behaviours is used to keep track of the type of channels created; unlike previous approaches by some of the authors no attempt is made to model any causality among the individual behaviours. Finally, we show that the system is a "conservative extension" of the usual type system for Standard ML.

The formal demonstration of semantic soundness, as well as the construction of the inference algorithm, are dealt with in companion papers [1, 13].

2 Inference System

The fragment of Concurrent ML [14] we have chosen for illustrating our approach has *expressions* ($e \in \mathit{Exp}$) and *constants* ($c \in \mathit{Con}$) given by the following syntax:

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid \mathtt{fn}\ x \Rightarrow e \mid e_1\, e_2 \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \\
 & \mid & \mathtt{rec}\ f\ x \Rightarrow e \mid \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \\
c & ::= & \mathtt{()} \mid \mathtt{true} \mid \mathtt{false} \mid n \mid \mathtt{+} \mid \mathtt{*} \mid \mathtt{=} \mid \cdots \\
 & \mid & \mathtt{pair} \mid \mathtt{fst} \mid \mathtt{snd} \mid \mathtt{nil} \mid \mathtt{cons} \mid \mathtt{hd} \mid \mathtt{tl} \mid \mathtt{isnil} \\
 & \mid & \mathtt{send} \mid \mathtt{receive} \mid \mathtt{sync} \mid \mathtt{channel} \mid \mathtt{fork}
\end{array}
$$

For expressions this includes constants, identifiers, function abstraction, application, polymorphic `let`-expressions, recursive functions, and conditionals; a *program* is an expression without any free identifiers.

Constants can be divided into four classes, according to whether they are *sequential* or *non-sequential* and according to whether they are *constructors* or *base functions*.

The sequential constructors include the unique element of the unit type, the two booleans, numbers ($n \in \mathit{Num}$), `pair` for constructing pairs, and `nil` and `cons` for constructing lists.

The sequential base functions include a selection of arithmetic operations, `fst` and `snd` for decomposing a pair, and `hd`, `tl` and `isnil` for decomposing and inspecting a list.

We shall allow to write (e_1,e_2) for $\texttt{pair}\,e_1\,e_2$, to write `[]` for `nil` and $[e_1 \cdots e_n]$ for $\texttt{cons}\,(e_1,\texttt{cons}(\cdots,\texttt{nil})\cdots)$, and to write $e_1;e_2$ for $\texttt{snd}\,(e_1,e_2)$ as this is a more readable way of expressing the sequencing between e_1 and e_2.

The unique flavour of Concurrent ML is due to the non-sequential constants which are the primitives for communication; we include five of these but more (in particular `choose` and `wrap`) can be added. The non-sequential constructors are `send` and `receive`: rather than actually enabling a communication they create *delayed communications* which are first-class entities that can be passed around freely. This leads to a very powerful programming discipline (in particular in the presence of `choose` and `wrap`) as is discussed in [14]. The non-sequential base functions are `sync`, `channel`, `fork` and these are explained below.

The function `sync` synchronises a delayed communication. Thus one process can send the value of e to another process by the expression $\texttt{sync}\,(\texttt{send}\,(\texttt{ch},e))$ where communication takes place along the channel `ch`. Similarly a process can receive a value from another process by the expression `sync (receive (ch))`.

The function `channel` allocates a new typed channel for communication when applied to `()`.

The function `fork` forks a new process e when applied to the expression $\texttt{fn}\ \text{dummy} \Rightarrow e$; this process will then execute concurrently with the other processes, one of which is the program itself.

Remark. We stated in the Introduction that our development is widely applicable. To this end it is worth pointing out the similarities between the `ref`-types of Standard ML and the delayed communications of Concurrent ML. In particular $\texttt{ref}\ e$ corresponds to `channel ()`, $e_1\texttt{:=}e_2$ corresponds to $\texttt{sync}\,(\texttt{send}\,(e_1,e_2))$, and $\texttt{!}e$ corresponds to $\texttt{sync}\,(\texttt{receive}\,e)$. Looking slightly ahead the Standard ML type t `ref` will correspond to the Concurrent ML type t `chan`. □

Example 2.1 Consider the program

```
fn f => let id = fn y =>
                   (if true
                    then f
                    else fn x =>
                           (sync (send (channel (), y));
                           x));
                   y
        in id id
```

that takes a function `f` as argument, defines an identity function `id`, and then applies `id` to itself. The identity function contains a conditional whose sole purpose is to force `f` and a locally defined function to have the same type. The locally defined function is yet another identity function except that it attempts to send the argument to `id` over a newly created channel. (To be able to execute one would need to fork a process that could read over the same channel.)

This program is of interest because it will be rejected in the subeffecting approach of [16] whereas it will be accepted in the system of [18]. We shall see that we will be able to type this program in our system as well! □

2.1 Annotated Types

To prepare for the type inference system we must clarify the syntax of types, effects, type schemes, and constraints. The syntax of *types* ($t \in \mathit{Typ}$) is given by:

$$\begin{array}{lll} t & ::= & \alpha \mid \texttt{unit} \mid \texttt{int} \mid \texttt{bool} \mid t_1 \times t_2 \mid t\ \texttt{list} \\ & \mid & t_1 \rightarrow^b t_2 \mid t\ \texttt{chan} \mid t\ \texttt{com}\ b \end{array}$$

Here we have base types for the unit type, booleans and integers; type variables are denoted α; composite types includes the product type, the function type and the list type; finally we have the type t `chan` for a typed channel allowing values of type t to be transmitted, and the type t `com` b for a delayed communication that will eventually result in a value of type t.

Except for the presence of a b-component in $t_1 \rightarrow^b t_2$ and t `com` b this is much the same type structure that is actually used in Concurrent ML [14]. The role of the b-component is to express the dynamic effect that takes place when the function is applied or the delayed communication synchronised. Motivated by [16] and (a simplified version of) [12] the syntax of *effects*, or *behaviours*, ($b \in \mathit{Beh}$) is given by:

$$b \quad ::= \quad \{t\ \text{CHAN}\} \mid \beta \mid \emptyset \mid b_1 \cup b_2$$

Here $\{t\ \text{CHAN}\}$ records the allocation of a channel of type t `chan`; behaviour variables are denoted β; $\emptyset$ denotes the minimal behaviour and $b_1 \cup b_2$ denotes the union of the two behaviours b_1 and b_2. The definition of types and behaviours is of course mutually recursive.

A *constraint set* C is a finite set of type $(t_1 \subseteq t_2)$ and behaviour inclusions $(b_1 \subseteq b_2)$. A *type scheme* ($ts \in TSch$) is given by

$$ts \quad ::= \quad \forall(\vec{\alpha}\vec{\beta} : C).\ t$$

where $\vec{\alpha}\vec{\beta}$ is the list of quantified type and behaviour variables, C is a constraint set, and t is the type. We regard type schemes as equivalent up to alpha-renaming of bound variables. There is a natural injection from types into type schemes which takes the type t into the type scheme $\forall(() : \emptyset).\ t$.

We list in Figure 1 the type schemes of a few selected constants. For those constants also to be found in Standard ML the constraint set is empty and the type is as in Standard ML except that the empty behaviour has been placed on all function types. The type of `sync` interacts closely with the types of `send` and `receive`: if `ch` is a channel of type t `chan`, the expression `receive ch` is going to have type t `com` $\emptyset$, and the expression `sync (receive ch)` is going to have type t; similarly for `send`. The type of `channel` clearly records the type of the created channel in the behaviour labelling the function type. Finally[1] the type of `fork` indicates that the argument may have any behaviour whatsoever, in particular this means that e in `fork (fn` dummy $\Rightarrow e$) is free to create new channels.

Following the approach of [15, 6] we will incorporate the effects of [16, 12] by defining a type inference system with judgements of the form

$$C, A \vdash e\ :\ \sigma\ \&\ b$$

where C is a constraint set, A is an environment i.e. a list $[x_1 : \sigma_1, \cdots, x_n : \sigma_n]$ of typing assumptions for identifiers, σ is a type t or a type scheme ts, and b is an effect. This means that e has type or type scheme σ, and that its execution will result in a behaviour described by b, assuming that free identifiers have types as specified by A and that all type and behaviour variables are related as described by C.

The overall structure of the type inference system of Figure 2 is very close to those of [15, 6] with a few components from [16, 12] thrown in; the novel ideas of our

[1] As discussed previously one might add `wrap` to the language: this constant transforms delayed communications of type t `com` b into delayed communications of type t' `com` b'; here b' (and thus also b) may be non-trivial.

c	TypeOf(c)
`+`	$\texttt{int} \times \texttt{int} \to^{\emptyset} \texttt{int}$
`pair`	$\forall(\alpha_1\alpha_2 : \emptyset).\ \alpha_1 \to^{\emptyset} \alpha_2 \to^{\emptyset} \alpha_1 \times \alpha_2$
`fst`	$\forall(\alpha_1\alpha_2 : \emptyset).\ \alpha_1 \times \alpha_2 \to^{\emptyset} \alpha_1$
`snd`	$\forall(\alpha_1\alpha_2 : \emptyset).\ \alpha_1 \times \alpha_2 \to^{\emptyset} \alpha_2$
`send`	$\forall(\alpha : \emptyset).\ (\alpha\ \texttt{chan}) \times \alpha \to^{\emptyset} (\alpha\ \texttt{com}\ \emptyset)$
`receive`	$\forall(\alpha : \emptyset).\ (\alpha\ \texttt{chan}) \to^{\emptyset} (\alpha\ \texttt{com}\ \emptyset)$
`sync`	$\forall(\alpha\beta : \emptyset).\ (\alpha\ \texttt{com}\ \beta) \to^{\beta} \alpha$
`channel`	$\forall(\alpha\beta : \{\{\alpha\ \text{CHAN}\} \subseteq \beta\}).\ \texttt{unit} \to^{\beta} (\alpha\ \texttt{chan})$
`fork`	$\forall(\alpha\beta : \emptyset).\ (\texttt{unit} \to^{\beta} \alpha) \to^{\emptyset} \texttt{unit}$

Figure 1: Type schemes for selected constants.

approach only show up as carefully constructed side conditions for some of the rules. Concentrating on the "overall picture" we thus have rather straightforward axioms for constants and identifiers; here $A(x)$ denotes the rightmost entry for x in A. The rules for abstraction and application are as usual in effect systems: the latent behaviour of the body of a function abstraction is placed on the arrow of the function type, and once the function is applied the latent behaviour is added to the effect of evaluating the function and its argument. The rule for `let` is straightforward given that both the `let`-bound expression and the body needs to be evaluated. The rule for recursion makes use of function abstraction to concisely represent the "fixed point requirement" of typing recursive functions; note that we do not admit polymorphic recursion. The rule for conditional is unable to keep track of which branch is chosen, therefore an upper approximation of the branches is taken. We then have separate rules for subtyping, instantiation and generalisation and we shall explain their side conditions shortly.

2.2 Subtyping

Rule (sub) generalises the subeffecting rule of [16] by incorporating subtyping and extends the subtyping rule of [15] to deal with effects. To do this we associate two kinds of judgements with a constraint set: the relations $C \vdash b_1 \subseteq b_2$ and $C \vdash t_1 \subseteq t_2$ are defined by the rules and axioms of Figure 3.

In all cases we write $\equiv$ for the equivalence induced by the orderings. We shall also write $C \vdash C'$ to mean that $C \vdash b_1 \subseteq b_2$ for all $(b_1 \subseteq b_2)$ in C' and that $C \vdash t_1 \subseteq t_2$ for all $(t_1 \subseteq t_2)$ in C'.

(con) $C, A \vdash c \,:\, \text{TypeOf(c)} \,\&\, \emptyset$

(id) $C, A \vdash x \,:\, A(x) \,\&\, \emptyset$

(abs) $$\frac{C, A[x : t_1] \vdash e \,:\, t_2 \,\&\, b}{C, A \vdash \texttt{fn}\ x \Rightarrow e \,:\, (t_1 \rightarrow^b\ t_2) \,\&\, \emptyset}$$

(app) $$\frac{C_1, A \vdash e_1 \,:\, (t_2 \rightarrow^b\ t_1) \,\&\, b_1 \qquad C_2, A \vdash e_2 \,:\, t_2 \,\&\, b_2}{(C_1 \cup C_2), A \vdash e_1\, e_2 \,:\, t_1 \,\&\, (b_1 \cup b_2 \cup b)}$$

(let) $$\frac{C_1, A \vdash e_1 \,:\, ts_1 \,\&\, b_1 \qquad C_2, A[x : ts_1] \vdash e_2 \,:\, t_2 \,\&\, b_2}{(C_1 \cup C_2), A \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \,:\, t_2 \,\&\, (b_1 \cup b_2)}$$

(rec) $$\frac{C, A[f : t] \vdash \texttt{fn}\ x \Rightarrow e \,:\, t \,\&\, b}{C, A \vdash \texttt{rec}\ f\ x \Rightarrow e \,:\, t \,\&\, b}$$

(if) $$\frac{C_0, A \vdash e_0 \,:\, \texttt{bool} \,\&\, b_0 \qquad C_1, A \vdash e_1 \,:\, t \,\&\, b_1 \qquad C_2, A \vdash e_2 \,:\, t \,\&\, b_2}{(C_0 \cup C_1 \cup C_2), A \vdash \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \,:\, t \,\&\, (b_0 \cup b_1 \cup b_2)}$$

(sub) $$\frac{C, A \vdash e \,:\, t \,\&\, b}{C, A \vdash e \,:\, t' \,\&\, b'}$$ if $C \vdash t \subseteq t'$ and $C \vdash b \subseteq b'$

(ins) $$\frac{C, A \vdash e \,:\, \forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0 \,\&\, b}{C, A \vdash e \,:\, S_0\, t_0 \,\&\, b}$$ if $\forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0$ is solvable from C by S_0

(gen) $$\frac{C \cup C_0, A \vdash e \,:\, t_0 \,\&\, b}{C, A \vdash e \,:\, \forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0 \,\&\, b}$$ if $\forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0$ is both well-formed, solvable from C, and satisfies $\{\vec{\alpha}\vec{\beta}\} \cap FV(C, A, b) = \emptyset$

Figure 2: The type inference system.

Ordering on behaviours

$$\text{(axiom)}\quad C \vdash b_1 \subseteq b_2 \qquad \text{if } (b_1 \subseteq b_2) \in C$$

$$\text{(refl)}\quad C \vdash b \subseteq b$$

$$\text{(trans)}\quad \frac{C \vdash b_1 \subseteq b_2 \quad C \vdash b_2 \subseteq b_3}{C \vdash b_1 \subseteq b_3}$$

$$(\text{CHAN})\quad \frac{C \vdash t \equiv t'}{C \vdash \{t \text{ CHAN}\} \subseteq \{t' \text{ CHAN}\}}$$

$$(\emptyset)\quad C \vdash \emptyset \subseteq b$$

$$(\cup)\quad C \vdash b_i \subseteq (b_1 \cup b_2) \qquad \text{for } i = 1, 2$$

$$\text{(lub)}\quad \frac{C \vdash b_1 \subseteq b \quad C \vdash b_2 \subseteq b}{C \vdash (b_1 \cup b_2) \subseteq b}$$

Ordering on types

$$\text{(axiom)}\quad C \vdash t_1 \subseteq t_2 \qquad \text{if } (t_1 \subseteq t_2) \in C$$

$$\text{(refl)}\quad C \vdash t \subseteq t$$

$$\text{(trans)}\quad \frac{C \vdash t_1 \subseteq t_2 \quad C \vdash t_2 \subseteq t_3}{C \vdash t_1 \subseteq t_3}$$

$$(\rightarrow)\quad \frac{C \vdash t_1' \subseteq t_1 \quad C \vdash t_2 \subseteq t_2' \quad C \vdash b \subseteq b'}{C \vdash (t_1 \rightarrow^{b} t_2) \subseteq (t_1' \rightarrow^{b'} t_2')}$$

$$(\times)\quad \frac{C \vdash t_1 \subseteq t_1' \quad C \vdash t_2 \subseteq t_2'}{C \vdash (t_1 \times t_2) \subseteq (t_1' \times t_2')}$$

$$(\texttt{list})\quad \frac{C \vdash t \subseteq t'}{C \vdash (t\ \texttt{list}) \subseteq (t'\ \texttt{list})}$$

$$(\texttt{chan})\quad \frac{C \vdash t \equiv t'}{C \vdash (t\ \texttt{chan}) \subseteq (t'\ \texttt{chan})}$$

$$(\texttt{com})\quad \frac{C \vdash t \subseteq t' \quad C \vdash b \subseteq b'}{C \vdash (t\ \texttt{com}\ b) \subseteq (t'\ \texttt{com}\ b')}$$

Figure 3: Subtyping and subeffecting.

The definition of $C \vdash b_1 \subseteq b_2$ is a fairly straightforward axiomatisation of set inclusion upon behaviours that are themselves sets of elements of the form t CHAN, with variables ranging over behaviours and with union and empty set; note that the premise for $C \vdash \{t_1 \text{ CHAN}\} \subseteq \{t_2 \text{ CHAN}\}$ is that $C \vdash t_1 \equiv t_2$.

The relation $C \vdash t_1 \subseteq t_2$ expresses the usual notion of subtyping, in particular it is contravariant in the argument position of a function type. In the case of `chan` note that the type t of t `chan` essentially occurs covariantly (when used in `receive`) and contravariantly (when used in `send`) at the same time; hence we must require that $t \equiv t'$ in order for t `chan` $\subseteq t'$ `chan` to hold.

2.3 Generalisation

We now explain some of the side conditions for the rules (ins) and (gen). This involves the notion of substitution: a mapping from type variables to types and from behaviour variables to behaviours[2] such that the domain is finite. Here the domain of a substitution S is $Dom(S) = \{\gamma \mid S\,\gamma \neq \gamma\}$ and the range is $Ran(S) = \bigcup \{FV(S\,\gamma) \mid \gamma \in Dom(S)\}$ where the concept of free variables, denoted $FV(\cdots)$, is standard. The identity substitution is denoted Id and we sometimes write $Inv(S) = Dom(S) \cup Ran(S)$ for the set of variables that are involved in the substitution S.

Rule (ins) is much as in [15] and merely says that to take an instance of a type scheme we must ensure that the constraints are satisfied; this is expressed using the notion of *solvability*:

Definition 2.2 The type scheme $\forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0$ is *solvable* from C by the substitution S_0 if $Dom(S_0) \subseteq \{\vec{\alpha}\vec{\beta}\}$ and if $C \vdash S_0\, C_0$.

Except for the well-formedness requirement (explained later), rule (gen) seems close to the corresponding rule in [15]: clearly we cannot generalise over variables free in the global type assumptions or global constraint sets, and as in effect systems (e.g. [16]) we cannot generalise over variables visible in the effect. Furthermore, as in [15] solvability is imposed to ensure that we do not create type schemes that have no instances; this condition ensures that the expressions `let x =` e_1 `in` e_2 and `let x =` e_1 `in x;`e_2 are going to be equivalent in the type system.

Example 2.3 Without an additional notion of well-formedness this does not give a semantically sound rule (gen); as an example consider the expression e given by

[2] We use γ to range over α's and β's as appropriate and use g range over t's and b's as appropriate.

```
let ch = channel ()
in  ···
    (sync(send(ch,7)))
    (sync(send(ch,true)))
```

and note that it is semantically unsound (at least if "$\cdots$" forked some process receiving twice over `ch` and adding the results). Writing $C = \{\{\alpha\ \text{CHAN}\} \subseteq \beta, \{\texttt{int}\ \text{CHAN}\} \subseteq \beta, \{\texttt{bool}\ \text{CHAN}\} \subseteq \beta\}$ and $C' = \{\{\alpha'\ \text{CHAN}\} \subseteq \beta\}$ then gives

$$C \cup C', [\,] \vdash \texttt{channel ()} : \alpha'\ \texttt{chan} \,\&\, \beta$$

and, without taking well-formedness into account, rule (gen) would give

$$C, [\,] \vdash \texttt{channel ()} : (\forall(\alpha' : C').\ \alpha'\ \texttt{chan}) \,\&\, \beta$$

because $\alpha' \notin FV(C, \beta)$ and $\forall(\alpha' : C').\ \alpha'\ \texttt{chan}$ is solvable from C by either of the substitutions $[\alpha' \mapsto \alpha]$, $[\alpha' \mapsto \texttt{int}]$ and $[\alpha' \mapsto \texttt{bool}]$. This then would give

$$C, [\texttt{ch} : \forall(\alpha' : C').\ \alpha'\ \texttt{chan}] \vdash \texttt{ch} : \texttt{int chan} \,\&\, \emptyset$$
$$C, [\texttt{ch} : \forall(\alpha' : C').\ \alpha'\ \texttt{chan}] \vdash \texttt{ch} : \texttt{bool chan} \,\&\, \emptyset$$

so that

$$C, [\,] \vdash e : t \,\&\, b$$

for suitable t and b. As it is easy to find S such that $\emptyset \vdash S\,C$, we shall see (by Lemma 2.15 and Lemma 2.16) that we even have

$$\emptyset, [\,] \vdash e : t' \,\&\, b'$$

for suitable t' and b'. This shows that some notion of well-formedness is essential for semantic soundness. □

The arrow relation

In order to formalise the notion of well-formedness we next associate a third kind of judgement and three kinds of closure with a constraint set.

Definition 2.4 The judgement $C \vdash \gamma_1 \leftarrow \gamma_2$ holds if there exists $(g_1 \subseteq g_2)$ in C such that $\gamma_i \in FV(g_i)$ for $i = 1, 2$.

The following trivial result proves useful:

Fact 2.5 Suppose $C \cup C_0 \vdash \gamma_1 \leftarrow \gamma_2$ with $\gamma_1 \notin FV(C)$; then $C_0 \vdash \gamma_1 \leftarrow \gamma_2$.

From this relation we define a number of other relations: $\rightarrow$ is the inverse of $\leftarrow$, i.e. $C \vdash \gamma_1 \rightarrow \gamma_2$ holds iff $C \vdash \gamma_2 \leftarrow \gamma_1$ holds, and $\leftrightarrow$ is the *union* of $\leftarrow$ and $\rightarrow$, i.e. $C \vdash \gamma_1 \leftrightarrow \gamma_2$ holds iff either $C \vdash \gamma_1 \leftarrow \gamma_2$ or $C \vdash \gamma_1 \rightarrow \gamma_2$ holds. As usual $\leftarrow^*$ (respectively $\rightarrow^*$, $\leftrightarrow^*$) denotes the reflexive and transitive closure of the relation.

For a set X of variables we then define the downwards closure $X^{C\downarrow}$, the upwards closure $X^{C\uparrow}$ and the bidirectional closure $X^{C\updownarrow}$ by:

$$\begin{aligned}
X^{C\downarrow} &= \{\gamma_1 \mid \exists \gamma_2 \in X : C \vdash \gamma_1 \leftarrow^* \gamma_2\} \\
X^{C\uparrow} &= \{\gamma_1 \mid \exists \gamma_2 \in X : C \vdash \gamma_1 \rightarrow^* \gamma_2\} \\
X^{C\updownarrow} &= \{\gamma_1 \mid \exists \gamma_2 \in X : C \vdash \gamma_1 \leftrightarrow^* \gamma_2\}
\end{aligned}$$

It is instructive to think of $C \vdash \gamma_1 \leftarrow \gamma_2$ as defining a directed graph structure upon $FV(C)$; then $X^{C\downarrow}$ is the reachability closure of X, $X^{C\uparrow}$ is the reachability closure in the graph where all edges are reversed, and $X^{C\updownarrow}$ is the reachability closure in the corresponding undirected graph.

Well-formedness

We can now define the notion of well-formedness for constraints and for type schemes; for the latter we make use of the arrow relations defined above.

Definition 2.6 *Well-formed constraint sets*

A constraint set C is *well-formed* if all right hand sides of $(g_1 \subseteq g_2)$ in C have g_2 to be a variable; in other words all inclusions of C have the form $t \subseteq \alpha$ or $b \subseteq \beta$.

The well-formedness assumption on constraint sets is motivated by the desire to be able to use the subtyping rules "backwards" (as spelled out in Lemma 2.7 below) and in ensuring that subtyping interacts well with the arrow relations (see Lemma 2.8 below).

Lemma 2.7 Suppose C is well-formed and that $C \vdash t \subseteq t'$.

- If $t' = t'_1 \rightarrow^{b'} t'_2$ there exist t_1, t_2 and b such that $t = t_1 \rightarrow^{b} t_2$ and such that $C \vdash t'_1 \subseteq t_1$, $C \vdash t_2 \subseteq t'_2$ and $C \vdash b \subseteq b'$.
- If $t' = t'_1\ \texttt{com}\ b'$ there exist t_1 and b such that $t = t_1\ \texttt{com}\ b$ and such that $C \vdash t_1 \subseteq t'_1$ and $C \vdash b \subseteq b'$.
- If $t' = t'_1 \times t'_2$ there exist t_1 and t_2 such that $t = t_1 \times t_2$ and such that $C \vdash t_1 \subseteq t'_1$ and $C \vdash t_2 \subseteq t'_2$.

- If $t' = t'_1$ `chan` there exist t_1 such that $t = t_1$ `chan` and such that $C \vdash t_1 \subseteq t'_1$ and $C \vdash t'_1 \subseteq t_1$.
- If $t' = t'_1$ `list` there exist t_1 such that $t = t_1$ `list` and such that $C \vdash t_1 \subseteq t'_1$.
- If $t' =$ `int` (respectively `bool`, `unit`) then $t =$ `int` (respectively `bool`, `unit`).

Proof See Appendix A. $\square$

Lemma 2.8 Suppose C is well-formed:

$$\text{if } C \vdash b \subseteq b' \text{ then } FV(b)^{C\downarrow} \subseteq FV(b')^{C\downarrow}.$$

Proof See Appendix A. $\square$

We now turn to well-formedness of type schemes where we ensure that the embedded constraints are themselves well-formed. Additionally we shall wish to ensure that the set of variables over which we generalise, is sensibly related to the constraints (unlike what was the case in Example 2.3). The key idea is that we do not generalise over γ_1 if $\gamma_1 \leftarrow \gamma_2$ and we are prevented from also generalising over γ_2. These considerations lead to:

Definition 2.9 *Well-formed type schemes*

A type scheme $\forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0$ is *well-formed* if C_0 is well-formed, if all $(g \subseteq \gamma)$ in C_0 contain at least one variable among $\{\vec{\alpha}\vec{\beta}\}$, and if $\{\vec{\alpha}\vec{\beta}\} = \{\vec{\alpha}\vec{\beta}\}^{C_0\uparrow}$.

It is essential for our development that the following property holds:

Fact 2.10 *Well-formedness and Substitutions*

If $\forall(\vec{\alpha}\vec{\beta} : C).\ t$ is well-formed then also $S\,(\forall(\vec{\alpha}\vec{\beta} : C).\ t)$ is well-formed (for all substitutions S).

Proof We can, without loss of generality, assume that $(Dom(S) \cup Ran(S)) \cap \{\vec{\alpha}\vec{\beta}\} = \emptyset$. Then $S\,(\forall(\vec{\alpha}\vec{\beta} : C).\ t) = \forall(\vec{\alpha}\vec{\beta} : S\,C).\ S\,t$. Consider $(g'_1 \subseteq g'_2)$ in $S\,C$; it is easy to see that it suffices to show that g'_2 is a variable in $\{\vec{\alpha}\vec{\beta}\}$.

Let $g'_1 = S\,g_1$ and $g'_2 = S\,g_2$ where $(g_1 \subseteq g_2) \in C$. Since C is well-formed it holds that g_2 is a variable, and since $FV(g_1, g_2) \cap \{\vec{\alpha}\vec{\beta}\} \neq \emptyset$ and since $\{\vec{\alpha}\vec{\beta}\} = \{\vec{\alpha}\vec{\beta}\}^{C\uparrow}$ it holds that $g_2 \in \{\vec{\alpha}\vec{\beta}\}$. Therefore $g'_2 = S\,g_2 = g_2$ so g'_2 is a variable in $\{\vec{\alpha}\vec{\beta}\}$. $\square$

Example 2.11 Continuing Example 2.3 note that $\{\alpha'\}^{C'\uparrow} = \{\alpha', \beta\}$ showing that our current notion of well-formedness prevents the erroneous typing. $\square$

Example 2.12 Continuing Example 2.1 we shall now briefly explain why it is accepted by our system. For this let us assume that `y` will have type α_y and that `x` will have type α_x. Then the locally defined function

```
fn x => (sync (send (channel (), y)); x)
```

will have type $\alpha_x \rightarrow^b \alpha_x$ for $b = \{\alpha_y \text{ CHAN}\}$. Due to our rule for subtyping we may let `f` have the type $\alpha_x \rightarrow^\emptyset \alpha_x$ and still be able to type the conditional. Clearly the expression defining `id` may be given the type $\alpha_y \rightarrow^\emptyset \alpha_y$ and the effect $\emptyset$. Since α_y is not free in the type of `f` we may use generalisation to give `id` the type scheme $\forall(\alpha_y : \emptyset).\ \alpha_y \rightarrow^\emptyset \alpha_y$. This then suffices for typing the application of `id` to itself.

The approach of [16] lacks subtyping although it has subeffecting. Consequently for the type of `f` to match that of the locally defined function we have to give `f` the type $\alpha_x \rightarrow^b \alpha_x$ where $b = \{\alpha_y \text{ CHAN}\}$. This then means that while the defining expression for `id` still has the type $\alpha_y \rightarrow^\emptyset \alpha_y$ we are unable to generalise it to $\forall(\alpha_y : \emptyset).\ \alpha_y \rightarrow^\emptyset \alpha_y$ because α_y is now free in the type of `f`. Consequently the application of `id` to itself cannot be typed. (It is interesting to point out that if one changed the applied occurrence of `f` in the program to the expression `fn z => f z` then subeffecting would suffice for generalising over α_y and hence would allow to type the self-application of `id`.)

We should also point out that in the approach of [18] one can generalise over α_y as well and hence type the self-application of `id` to itself. To see this, first note that α_y is classified as an imperative type variable (rather than an applicative type variable which would directly have allowed the generalisation) because α_y is used in the channel construct and thus has a side effect. Despite of this, next note that defining expression for the `id` function is classified as non-expansive (rather as expansive which would directly have prohibited the generalisation of imperative type variables) because all side effects occurring in the definition of `id` are "protected" by a function abstraction and hence not "dangerous". We refer to [18] for the details. □

2.4 Properties of the Inference System

We now list a few basic properties of the inference system that we shall use later.

Fact 2.13 For all constants c of Figure 1, the type scheme $\text{TypeOf}(c)$ is closed, well-formed and solvable from $\emptyset$.

Fact 2.14 *Solvability and Well-formedness of Typing Judgements*

If $C, A \vdash e : \sigma \,\&\, b$ and A is well-formed and solvable from C then σ is well-formed and solvable from C.

Proof A straightforward induction on the shape of the inference tree; for constants we make use of Fact 2.13. □

Lemma 2.15 *Substitution Lemma*

For all substitutions S:

(a) If $C \vdash C'$ then $S\,C \vdash S\,C'$.

(b) If $C, A \vdash e : \sigma \,\&\, b$ then $S\,C, S\,A \vdash e : S\,\sigma \,\&\, S\,b$ (and has the same shape).

Proof See Appendix A. □

Lemma 2.16 *Entailment Lemma*

For all sets C' of constraints satisfying $C' \vdash C$:

(a) If $C \vdash C_0$ then $C' \vdash C_0$;

(b) If $C, A \vdash e : \sigma \,\&\, b$ then $C', A \vdash e : \sigma \,\&\, b$ (and has the same shape).

Proof See Appendix A. □

Fact 2.17 Let x and y be distinct identifiers: if $C, A_1[x : \sigma_1][y : \sigma_2]A_2 \vdash e : \sigma \,\&\, b$ then $C, A_1[y : \sigma_2][x : \sigma_1]A_2 \vdash e : \sigma \,\&\, b$ (and has the same shape).

Fact 2.18 Let x be an identifier not occurring in e and let t be an arbitrary type; if $C, A \vdash e : \sigma \,\&\, b$ then $C, A[x : t] \vdash e : \sigma \,\&\, b$ (and has the same shape).

Proof Let α be a fresh type variable. Then a straight-forward induction in the proof tree (using Fact 2.17) tells us that $C, A[x : \alpha] \vdash e : \sigma \,\&\, b$ (and has the same shape). Now apply Lemma 2.15 with the substitution $[\alpha \mapsto t]$. □

2.5 Proof Normalisation

It turns out that the proof of semantic soundness as well as the proof of completeness of an inference algorithm is complicated by the presence of the non-syntax directed rules (sub), (gen) and (ins) of Figure 2. This motivates trying to normalise general inference trees into a more manageable shape; to this end we define the notions of "normalised" and "strongly normalised" inference trees. But first we define an auxiliary concept:

Definition 2.19 *Constraint-Saturated*

An inference tree for $C, A \vdash e : \sigma \,\&\, b$ is constraint-saturated, written $C, A \vdash_c e : \sigma \,\&\, b$, if and only if all occurrences of the rules (app), (let), and (if) have the same constraints in their premises; in the notation of Figure 2 this means that $C_1 = C_2$ for (app) and (let) and that $C_0 = C_1 = C_2$ for (if).

Fact 2.20 *Enforcing Constraint-Saturation*

Given an inference tree for $C, A \vdash e : \sigma \,\&\, b$ there exists a constraint-saturated inference tree $C, A \vdash_c e : \sigma \,\&\, b$ (that has the same shape).

Proof A straightforward induction in the shape of the inference tree using Lemma 2.16 in the cases (app), (let) and (if). □

We now define the central concepts of T- and TS-normalised inference trees.

Definition 2.21 *Normalisation*

An inference tree for $C, A \vdash e : t \,\&\, b$ is *T-normalised* if it is created by:

- (con) or (id); or
- (ins) applied to (con) or (id); or
- (abs), (app), (rec), (if) or (sub) applied to T-normalised inference trees; or
- (let) applied to a TS-normalised inference tree and a T-normalised inference tree.

An inference tree for $C, A \vdash e : ts \,\&\, b$ is *TS-normalised* if it is created by:

- (gen) applied to a T-normalised inference tree.

We shall write $C, A \vdash_n e : \sigma \,\&\, b$ if the inference tree is T-normalised (if σ is a type) or TS-normalised (if σ is a type scheme).

Lemma 2.22 *Normalisation Lemma*

If A is well-formed and solvable from C then an inference tree $C, A \vdash e : \sigma \,\&\, b$ can be transformed into one $C, A \vdash_n e : \sigma \,\&\, b$ that is normalised.

Proof See Appendix A. □

A somewhat stronger property is the following:

Definition 2.23 *Strongly Normalised*

An inference tree for $C, A \vdash e : \sigma \,\&\, b$ is strongly normalised if it:

- is constraint-saturated; and
- is normalised; and
- has an occurrence of (sub) after each T-normalised inference tree in $C, A \vdash e : \sigma \,\&\, b$ not created by (sub); and
- has no consecutive applications of (sub).

We write $C, A \vdash_s e : \sigma \,\&\, b$ when this is the case.

Lemma 2.24 *Enforcing Strong Normalisation*

If A is well-formed and solvable from C then an inference tree $C, A \vdash e : \sigma \,\&\, b$ can be transformed into one $C, A \vdash_s e : \sigma \,\&\, b$ that is strongly normalised.

Proof By Lemma 2.22 we can obtain a normalised inference tree that by Fact 2.20 can be assumed to be constraint-saturated. Now after each T-normalised subinference insert a trivial application of (sub); this maintains the property of being normalised and constraint-saturated. Now use the transitivity of subtyping and subeffecting to contract all consecutive applications of (sub) into just one application; this maintains the property of being normalised and constraint-saturated. $\Box$

2.6 Conservative Extension

We finally show that our inference system is a conservative extension of the system for ML type inference. For this purpose we restrict ourselves to consider *sequential* expressions only, that is expressions without the non-sequential constants `channel`, `fork`, `sync`, `send`, and `receive`.

An ML type u (as opposed to a CML type t, in the following just denoted type) is either a type variable α, a base type like `int`, a function type $u_1 \to u_2$, a product type $u_1 \times u_2$, or a list type u_1 `list`. An ML type scheme is of the form $\forall \vec{\alpha}.u$.

We say that a type is *sequential* if it does not contain subtypes of form t `com` b or t `chan`. From a sequential type t we construct an ML type $\epsilon(t)$ as follows: $\epsilon(\alpha) = \alpha$, $\epsilon(\texttt{int}) = \texttt{int}$, $\epsilon(t_1 \to^b t_2) = \epsilon(t_1) \to \epsilon(t_2)$, $\epsilon(t_1 \times t_2) = \epsilon(t_1) \times \epsilon(t_2)$, and $\epsilon(t_1\ \texttt{list}) = \epsilon(t_1)\ \texttt{list}$. It is convenient also to define $\epsilon(t)$ for non-sequential types and we (somewhat arbitrarily) do this by stipulating $\epsilon(t\ \texttt{com}\ b) = \epsilon(t\ \texttt{chan}) = \epsilon(t)$.

$$\text{(con)}\quad A \vdash_{\mathrm{ML}} c : \mathrm{MLTypeOf}(c)$$

$$\text{(id)}\quad A \vdash_{\mathrm{ML}} x : A(x)$$

$$\text{(abs)}\quad \frac{A[x:u_1] \vdash_{\mathrm{ML}} e : u_2}{A \vdash_{\mathrm{ML}} \texttt{fn}\ x \Rightarrow e : u_1 \to u_2}$$

$$\text{(app)}\quad \frac{A \vdash_{\mathrm{ML}} e_1 : u_2 \to u_1,\ A \vdash_{\mathrm{ML}} e_2 : u_2}{A \vdash_{\mathrm{ML}} e_1\, e_2 : u_1}$$

$$\text{(let)}\quad \frac{A \vdash_{\mathrm{ML}} e_1 : us_1,\ A[x:us_1] \vdash_{\mathrm{ML}} e_2 : u_2}{A \vdash_{\mathrm{ML}} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : u_2}$$

$$\text{(ins)}\quad \frac{A \vdash_{\mathrm{ML}} e : \forall\vec{\alpha}.u}{A \vdash_{\mathrm{ML}} e : R\,u} \qquad \text{if } Dom(R) \subseteq \{\vec{\alpha}\}$$

$$\text{(gen)}\quad \frac{A \vdash_{\mathrm{ML}} e : u}{A \vdash_{\mathrm{ML}} e : \forall\vec{\alpha}.u} \qquad \text{if } FV(A) \cap \{\vec{\alpha}\} = \emptyset$$

Figure 4: The core of the ML type inference system.

We say that a type scheme $ts = \forall(\vec{\alpha}\,\vec{\beta} : C).\ t$ is *sequential* if C is empty and if t is sequential. From a sequential type scheme $ts = \forall(\vec{\alpha}\,\vec{\beta} : \emptyset).\ t$ we construct an ML type scheme $\epsilon(ts)$ as follows: $\epsilon(ts) = \forall\vec{\alpha}.\epsilon(t)$. (We shall dispense with defining $\epsilon(ts)$ on non-sequential type schemes for reasons to be discussed in Appendix A.)

The core of the ML type inference system is depicted in Figure 4. It employs a function MLTypeOf which to each sequential constant assigns either an ML type or an ML type scheme; as an example we have MLTypeOf(`pair`) $= \forall\alpha_1\alpha_2.\alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$.

Fact 2.25 For a sequential constant c we have that TypeOf(c) is sequential.

Assumption 2.26 For a sequential constant c we have that MLTypeOf(c) = ϵ(TypeOf(c)).

We are now ready to state that our system conservatively extends ML.

Theorem 2.27 Let e be a sequential expression. If $\emptyset \vdash_{\mathrm{ML}} e : u$ then there exists a sequential type t with $\epsilon(t) = u$ such that $\emptyset, \emptyset \vdash e : t \,\&\, \emptyset$; and if $\emptyset, \emptyset \vdash e : t \,\&\, b$ then there exists an ML type u with $\epsilon(t) = u$ such that $\emptyset \vdash_{\mathrm{ML}} e : u$.

Proof: See Appendix A.

3 Conclusion

We have extended previous work on integrating polymorphism, subtyping and effects into a combined annotated type and effect system. The development was illustrated for a fragment of Concurrent ML but is equally applicable to Standard ML with references. A main ingredient of the approach was the notion of constraint closure, in particular the notion of upwards closure. We hope that this system will provide a useful basis for developing a variety of program analyses; in particular closure, binding-time and communication analyses for languages with imperative or concurrent effects.

The system developed here includes no causality concerning the temporal order of effects; a future goal is to incorporate aspects of the causality information for the communication structure of Concurrent ML that was developed in [12]. Another (and harder) goal is to incorporate decidable fragments of polymorphic recursion. Finally, it should prove interesting to apply these ideas also to strongly typed languages with object-oriented features.

Acknowledgement This work has been supported in part by the *DART* project (Danish Natural Science Research Council) and the *LOMAPS* project (ESPRIT BRA project 8130); it represents joint work among the authors.

References

[1] T.Amtoft, F.Nielson, H.R.Nielson, J.Ammann: Polymorphic Subtypes for Effect Analysis: the Semantics, 1996.

[2] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. of POPL '82*. ACM Press, 1982.

[3] Y.-C. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *Proc. TAPSOFT '89*. SLNCS 352, 1989.

[4] Y.-C. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73, 1990.

[5] F. Henglein and C. Mossin. Polymorphic binding-time analysis. In *Proc. ESOP '94*, pages 287–301. SLNCS 788, 1994.

[6] M. P. Jones. A theory of qualified types. In *Proc. ESOP '92*, pages 287–306. SLNCS 582, 1992.

[7] X. Leroy and P. Weis. Polymorphic type inference and assignment. In *Proc. POPL '91*, pages 291–302. ACM Press, 1991.

[8] R. Milner. A theory of type polymorphism in programming. *Journal of Computer Systems*, 17:348–375, 1978.

[9] J. C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3), 1991.

[10] F. Nielson and H.R. Nielson. Constraints for polymorphic behaviours for Concurrent ML. In *Proc. CCL'94*. SLNCS 845, 1994.

[11] H.R. Nielson and F. Nielson. Automatic binding analysis for a typed λ-calculus. *Science of Computer Programming*, 10:139–176, 1988.

[12] H.R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *Proc. POPL'94*, pages 84–97. ACM Press, 1994.

[13] F.Nielson, H.R.Nielson, T.Amtoft: Polymorphic Subtypes for Effect Analysis: the Algorithm, 1996.

[14] J. H. Reppy. Concurrent ML: Design, application and semantics. In *Proc. Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198. SLNCS 693, 1993.

[15] G. S. Smith. Polymorphic inference with overloading and subtyping. In SLNCS 668, *Proc. TAPSOFT '93*, 1993. Also see: Principal Type Schemes for Functional Programs with Overloading and Subtyping: *Science of Computer Programming* 23, pp. 197-226, 1994.

[16] J. P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111, 1994.

[17] Y.-M. Tang. *Control Flow Analysis by Effect Systems and Abstract Interpretation.* PhD thesis, Ecoles des Mines de Paris, 1994.

[18] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.

[19] A. K. Wright. Typing references by effect inference. In *Proc. ESOP '92*, pages 473–491. SLNCS 582, 1992.

A Details of Proofs

Well-formedness

Lemma 2.7 Suppose C is well-formed and that $C \vdash t \subseteq t'$.

- If $t' = t'_1 \rightarrow^{b'} t'_2$ there exist t_1, t_2 and b such that $t = t_1 \rightarrow^{b} t_2$ and such that $C \vdash t'_1 \subseteq t_1$, $C \vdash t_2 \subseteq t'_2$ and $C \vdash b \subseteq b'$.
- If $t' = t'_1\ \mathtt{com}\ b'$ there exist t_1 and b such that $t = t_1\ \mathtt{com}\ b$ and such that $C \vdash t_1 \subseteq t'_1$ and $C \vdash b \subseteq b'$.
- If $t' = t'_1 \times t'_2$ there exist t_1 and t_2 such that $t = t_1 \times t_2$ and such that $C \vdash t_1 \subseteq t'_1$ and $C \vdash t_2 \subseteq t'_2$.
- If $t' = t'_1\ \mathtt{chan}$ there exist t_1 such that $t = t_1\ \mathtt{chan}$ and such that $C \vdash t_1 \subseteq t'_1$ and $C \vdash t'_1 \subseteq t_1$.
- If $t' = t'_1\ \mathtt{list}$ there exist t_1 such that $t = t_1\ \mathtt{list}$ and such that $C \vdash t_1 \subseteq t'_1$.
- If $t' = \mathtt{int}$ (respectively $\mathtt{bool}$, $\mathtt{unit}$) then $t = \mathtt{int}$ (respectively $\mathtt{bool}$, $\mathtt{unit}$).

In addition we are going to prove that the size of each of the latter inference trees is strictly less than the size of the inference tree for $C \vdash t \subseteq t'$. Here the size of an inference tree is defined as the number of (not necessarily different) symbols occurring in the tree, except that occurrences in C do not count.

Proof We only consider the case $t' = t'_1 \rightarrow^{b'} t'_2$, as the others are similar. The proof is carried out by induction in the inference tree, and since C is well-formed the last rule applied must be either (refl), (trans) or ($\rightarrow$).

(refl): the claim is trivial[3].

(trans): assume that $C \vdash t \subseteq t'$ by means of a tree of size n because $C \vdash t \subseteq t''$ by means of a tree of size n'' and because $C \vdash t'' \subseteq t'$ by means of a tree of size n'. Here $n = n' + n'' + |t| + |t'| + 2$. By applying the induction hypothesis on the latter inference we find t''_1, t''_2 and b'' such that $t'' = t''_1 \rightarrow^{b''} t''_2$ and such that $C \vdash t'_1 \subseteq t''_1$ and $C \vdash t''_2 \subseteq t'_2$ and $C \vdash b'' \subseteq b'$, each judgement by means of an inference tree of size $< n'$. By applying the induction hypothesis on the former inference ($C \vdash t \subseteq t''$) we find t_1, t_2 and b such that $t = t_1 \rightarrow^{b} t_2$ and such that $C \vdash t''_1 \subseteq t_1$ and $C \vdash t_2 \subseteq t''_2$ and $C \vdash b \subseteq b''$, each judgement by means of an inference tree of size $< n''$. We thus have $C \vdash t'_1 \subseteq t_1$, by means of an inference tree of size $< n' + n'' + |t'_1| + |t_1| + 2 < n' + n'' + |t'| + |t| + 2 = n$. By similar reasoning

[3] This case is the reason for not defining the size of a tree as the number of inferences.

we infer that $C \vdash t_2 \subseteq t_2'$ and $C \vdash b \subseteq b'$, each judgement by means of an inference tree of size $< n$.

($\rightarrow$): the claim is trivial. $\square$

For variables we need a different kind of lemma:

Lemma A.1 Suppose $C \vdash \alpha \subseteq \alpha'$ with C well-formed. Then $\alpha \in \{\alpha'\}^{C\downarrow}$.

Proof Induction in the proof tree, performing case analysis on the last rule applied:

axiom: then $(\alpha \subseteq \alpha') \in C$ so the claim is trivial.

refl: the claim is trivial.

trans: assume that $C \vdash \alpha \subseteq \alpha'$ because $C \vdash \alpha \subseteq t''$ and $C \vdash t'' \subseteq \alpha'$. By using Lemma 2.7 on the inference $C \vdash \alpha \subseteq t''$ we infer that t'' is a variable α''. By applying the induction hypothesis we infer that $\alpha \in \{\alpha''\}^{C\downarrow}$ and that $\alpha'' \in \{\alpha'\}^{C\downarrow}$, from which we conclude that $\alpha \in \{\alpha'\}^{C\downarrow}$. $\square$

Lemma 2.8 Suppose C is well-formed:

if $C \vdash b \subseteq b'$ then $FV(b)^{C\downarrow} \subseteq FV(b')^{C\downarrow}$, and
if $C \vdash t \equiv t'$ then $FV(t)^{C\downarrow} = FV(t')^{C\downarrow}$.

Proof Induction in the size of the inference tree, where we define the size of the inference tree for $C \vdash t \equiv t'$ as the sum of the size of the inference tree for $C \vdash t \subseteq t'$ and the size of the inference tree for $C \vdash t' \subseteq t$.

First we consider the part concerning behaviours, performing case analysis on the last inference rule applied:

(axiom): then $(b \subseteq b') \in C$ so since C is well-formed b' is a variable; hence the claim.

(refl): the claim is trivial.

(trans): assume that $C \vdash b \subseteq b'$ because $C \vdash b \subseteq b''$ and $C \vdash b'' \subseteq b'$. The induction hypothesis tells us that $FV(b)^{C\downarrow} \subseteq FV(b'')^{C\downarrow}$ and that $FV(b'')^{C\downarrow} \subseteq FV(b')^{C\downarrow}$; hence the claim.

(CHAN): assume that $C \vdash \{t \text{ CHAN}\} \subseteq \{t' \text{ CHAN}\}$ because $C \vdash t \equiv t'$. The induction hypothesis tells us that $FV(t)^{C\downarrow} = FV(t')^{C\downarrow}$; hence the claim.

($\emptyset$:) the claim is trivial.

($\cup$:) the claim is trivial.

(lub): assume that $C \vdash b_1 \cup b_2 \subseteq b'$ because $C \vdash b_1 \subseteq b'$ and $C \vdash b_2 \subseteq b'$. The induction hypothesis tells us that $FV(b_1)^{C\downarrow} \subseteq FV(b')^{C\downarrow}$ and that $FV(b_2)^{C\downarrow} \subseteq FV(b')^{C\downarrow}$, from which we infer that $FV(b_1 \cup b_2)^{C\downarrow} = FV(b_1)^{C\downarrow} \cup FV(b_2)^{C\downarrow} \subseteq FV(b')^{C\downarrow}$.

Next we consider the part concerning types, where we perform case analysis on the form of t':

$t' = t'_1 \rightarrow^{b'} t'_2$: Let n_1 be the size of the inference tree for $C \vdash t \subseteq t'$ and let n_2 be the size of the inference tree for $C \vdash t' \subseteq t$. Lemma 2.7 (applied to the former inference) tells us that there exist t_1, b and t_2 such that $t = t_1 \rightarrow^{b} t_2$ and such that $C \vdash t'_1 \subseteq t_1$, $C \vdash b \subseteq b'$ and $C \vdash t_2 \subseteq t'_2$, where each inference tree is of size $< n_1$ (due to the remark at the beginning of the proof). Lemma 2.7 (applied to the latter inference, i.e. $C \vdash t' \subseteq t$) tells us that $C \vdash t_1 \subseteq t'_1$, $C \vdash b' \subseteq b$ and $C \vdash t'_2 \subseteq t_2$, where each inference tree is of size $< n_2$.

Thus $C \vdash t_1 \equiv t'_1$ and $C \vdash t_2 \equiv t'_2$, where each inference tree has size $< n_1 + n_2$. We can thus apply the induction hypothesis to infer that $FV(t_1)^{C\downarrow} = FV(t'_1)^{C\downarrow}$ and that $FV(t_2)^{C\downarrow} = FV(t'_2)^{C\downarrow}$; and similarly we can infer that $FV(b)^{C\downarrow} \subseteq FV(b')^{C\downarrow}$ and that $FV(b')^{C\downarrow} \subseteq FV(b)^{C\downarrow}$. This enables us to concluce that $FV(t)^{C\downarrow} = FV(t')^{C\downarrow}$.

t' has a topmost type constructor other than $\rightarrow$: we can proceed as above.

t' is a variable: Since $C \vdash t' \subseteq t$ we can use Lemma 2.7 to infer that t is a variable; then we use Lemma A.1 to infer that $FV(t') \subseteq FV(t)^{C\downarrow}$. Similarly we can infer $FV(t) \subseteq FV(t')^{C\downarrow}$. This implies the desired relation $FV(t)^{C\downarrow} = FV(t')^{C\downarrow}$. □

Properties of the inference system

Lemma 2.15 For all substitutions S:

(a) If $C \vdash C'$ then $S\,C \vdash S\,C'$.

(b) If $C, A \vdash e : \sigma \;\&\; b$ then $S\,C, S\,A \vdash e : S\,\sigma \;\&\; S\,b$ (and has the same shape).

Proof The claim (a) is straight-forward by induction on the inference $C \vdash g_1 \subseteq g_2$ for each $(g_1 \subseteq g_2) \in C'$. For the claim (b) we proceed by induction on the inference.

For the case (con) we use that the type schemes of Table 1 are closed (Fact 2.13). For the case (id) the claim is immediate, and for the cases (abs), (app), (let), (rec), (if) it follows directly using the induction hypothesis. For the case (sub) we use (a) together with the induction hypothesis.

The case (ins). Then $C, A \vdash e : S_0\, t_0 \,\&\, b$ because $C, A \vdash e : \forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0 \,\&\, b$ where $C \vdash S_0\, C_0$ and $Dom(S_0) \subseteq \{\vec{\alpha}\vec{\beta}\}$, and wlog. we can assume that $\{\vec{\alpha}\vec{\beta}\}$ is disjoint from $Inv(S)$. The induction hypothesis gives

$$S\,C, S\,A \vdash e : \forall(\vec{\alpha}\vec{\beta} : S\,C_0).\ S\,t_0 \,\&\, S\,b. \tag{1}$$

From (a) we get $S\,C \vdash S\,S_0\,C_0$. Let $S_0' = [\vec{\alpha}\vec{\beta} \mapsto S\,S_0\,(\vec{\alpha}\vec{\beta})]$, then on $FV(t_0, C_0)$ it holds that $S_0'\,S = S\,S_0$. Therefore $S\,C \vdash S_0'\,S\,C_0$, so we can apply (ins) on (1) with S_0' as the "instance substitution" to get $S\,C, S\,A \vdash e : S_0'\,S\,t_0 \,\&\, S\,b$. Since $S_0'\,S\,t_0 = S\,S_0\,t_0$ this is the required result.

The case (gen). Then $C, A \vdash e : \forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0 \,\&\, b$ because $C \cup C_0, A \vdash e : t_0 \,\&\, b$, and

$$\forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0 \text{ is well-formed,} \tag{2}$$

$$\text{there exists } S_0 \text{ with } Dom(S_0) \subseteq \{\vec{\alpha}\vec{\beta}\} \text{ such that } C \vdash S_0\,C_0, \text{ and} \tag{3}$$

$$\{\vec{\alpha}\vec{\beta}\} \cap FV(C, A, b) = \emptyset \tag{4}$$

Define $R = [\vec{\alpha}\vec{\beta} \mapsto \vec{\alpha'}\vec{\beta'}]$ with $\{\vec{\alpha'}\vec{\beta'}\}$ fresh. We then apply the induction hypothesis (with $S\,R$) and due to (4) this gives us $S\,C \cup S\,R\,C_0, S\,A \vdash e : S\,R\,t_0 \,\&\, S\,b$. Below we prove

$$\forall(\vec{\alpha'}\vec{\beta'} : S\,R\,C_0).\ S\,R\,t_0 = S\,(\forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0) \text{ is well-formed,} \tag{5}$$

$$\text{there exists } S' \text{ with } Dom(S') \subseteq \{\vec{\alpha'}\vec{\beta'}\} \text{ such that } S\,C \vdash S'\,S\,R\,C_0, \text{ and} \tag{6}$$

$$\{\vec{\alpha'}\vec{\beta'}\} \cap FV(S\,C, S\,A, S\,b) = \emptyset \tag{7}$$

It then follows that $S\,C, S\,A \vdash e : S\,(\forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0) \,\&\, S\,b$ as required. Clearly the inference has the same shape.

First we observe that (5) follows from (2) and Fact 2.10. For (6) define $S' = [\vec{\alpha'}\vec{\beta'} \mapsto S\,S_0\,(\vec{\alpha}\vec{\beta})]$. From $C \vdash S_0\,C_0$ and (a) we get $S\,C \vdash S\,S_0\,C_0$. Since $S'\,S\,R = S\,S_0$ on $FV(C_0)$ the result follows. Finally (7) holds trivially by choice of $\vec{\alpha'}\vec{\beta'}$. □

Lemma 2.16 For all sets C' of constraints satisfying $C' \vdash C$:

(a) If $C \vdash C_0$ then $C' \vdash C_0$.

(b) If $C, A \vdash e : \sigma \,\&\, b$ then $C', A \vdash e : \sigma \,\&\, b$ (and has the same shape).

Proof The claim (a) is straight-forward by induction on the inference $C \vdash g_1 \subseteq g_2$ for each $(g_1 \subseteq g_2) \in C_0$. For the claim (b) we proceed by induction on the inference.

For the cases (con), (id) the claim is immediate, and for the cases (abs), (app), (let), (rec), (if) it follows directly using the induction hypothesis. For the case (sub) we use (a) together with the induction hypothesis.

The case (ins). Then $C, A \vdash e : S_0\, t_0 \,\&\, b$ because $C, A \vdash e : \forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0 \,\&\, b$ and $C \vdash S_0\, C_0$ and $Dom(S_0) \subseteq \{\vec{\alpha}\vec{\beta}\}$. The induction hypothesis gives $C', A \vdash e : \forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0 \,\&\, b$. From (a) we have $C' \vdash S_0\, C_0$ so $C', A \vdash e : S_0\, t_0 \,\&\, b$ follows. Clearly the inference has the same shape.

The case (gen). Then $C, A \vdash e : \forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0 \,\&\, b$ because $C \cup C_0, A \vdash e : t_0 \,\&\, b$ and

$$\forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0 \text{ is well-formed,} \tag{8}$$

$$\text{there exists } S \text{ with } Dom(S) \subseteq \{\vec{\alpha}\vec{\beta}\} \text{ such that } C \vdash S\, C_0, \text{ and} \tag{9}$$

$$\{\vec{\alpha}\vec{\beta}\} \cap FV(C, A, b) = \emptyset \tag{10}$$

We now use a small trick: let R be a renaming of the variables of $\{\vec{\alpha}\vec{\beta}\} \cap FV(C')$ to fresh variables. From $C' \vdash C$ and Lemma 2.15(a) we get $R\,C' \vdash R\,C$ and using (10) we get $R\,C = C$ so $R\,C' \vdash C$. Clearly $R\,C' \cup C_0 \vdash C \cup C_0$ so the induction hypothesis gives $R\,C' \cup C_0, A \vdash e : t_0 \,\&\, b$. Below we verify that

$$\text{there exists } S' \text{ with } Dom(S') \subseteq \{\vec{\alpha}\vec{\beta}\} \text{ such that } R\,C' \vdash S'\, C_0, \text{ and} \tag{11}$$

$$\{\vec{\alpha}\vec{\beta}\} \cap FV(R\,C', A, b) = \emptyset \tag{12}$$

and we then have $R\,C', A \vdash e : \forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0 \,\&\, b$. Now define the substitution R' such that $Dom(R') = Ran(R)$ and $R'\,\gamma' = \gamma$ if $R\,\gamma = \gamma'$ and $\gamma' \in Dom(R')$. Using Lemma 2.15(b) with the substitution R' we get $C', A \vdash e : \forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0 \,\&\, b$ as required. Clearly the inference has the same shape.

To prove (11) define $S' = S$. Above we showed that $R\,C' \vdash C$ so using (9) and (a) we get $R\,C' \vdash S'\, C_0$ as required. Finally (12) follows trivially from $\{\vec{\alpha}\vec{\beta}\} \cap FV(R\,C') = \emptyset$. $\square$

Proof normalisation

Lemma 2.22 If A is well-formed and solvable from C then an inference tree $C, A \vdash e : \sigma \,\&\, b$ can be transformed into one $C, A \vdash_n e : \sigma \,\&\, b$ that is normalised.

Proof Using Fact 2.20, we can, without loss of generality, assume that we have a constraint-saturated inference tree for $C, A \vdash e : \sigma \,\&\, b$. We proceed by induction on the inference.

The case (con). We assume $C, A \vdash_c c : \mathrm{TypeOf}(c) \,\&\, \emptyset$. If $\mathrm{TypeOf}(c)$ is a type then we already have a T-normalised inference. So assume $\mathrm{TypeOf}(c)$ is a type scheme $\forall(\vec{\alpha}\vec{\beta} : C_0).\, t_0$ and let R be a renaming of $\vec{\alpha}\vec{\beta}$ to fresh variables $\vec{\alpha'}\vec{\beta'}$. We can then construct the following TS-normalised inference tree:

$$\dfrac{\dfrac{\dfrac{}{C \cup R\,C_0, A \vdash c : \forall(\vec{\alpha}\vec{\beta} : C_0).\, t_0 \,\&\, \emptyset}\,(\mathrm{con})}{C \cup R\,C_0, A \vdash c : R\,t_0 \,\&\, \emptyset}\,(\mathrm{ins})}{C, A \vdash c : \forall(\vec{\alpha'}\vec{\beta'} : R\,C_0).\, R\,t_0 \,\&\, \emptyset}\,(\mathrm{gen})$$

The rule (ins) is applicable since $Dom(R) \subseteq \{\vec{\alpha}\vec{\beta}\}$ and $C \cup R\,C_0 \vdash R\,C_0$. The rule (gen) is applicable because $\forall(\vec{\alpha}\vec{\beta} : C_0).\, t_0 = \forall(\vec{\alpha'}\vec{\beta'} : R\,C_0).\, R\,t_0$ (up to alpha-renaming) is well-formed and solvable from C (Fact 2.13), and furthermore $\{\vec{\alpha'}\vec{\beta'}\} \cap FV(C, A, \emptyset) = \emptyset$ holds by choice of $\vec{\alpha'}\vec{\beta'}$.

The case (id). We assume $C, A \vdash_c x : A(x) \,\&\, \emptyset$. If $A(x)$ is a type then we already have a T-normalised inference. So assume $A(x) = \forall(\vec{\alpha}\vec{\beta} : C_0).\, t_0$ and let R be a renaming of $\vec{\alpha}\vec{\beta}$ to fresh variables $\vec{\alpha'}\vec{\beta'}$. We can then construct the following TS-normalised inference tree:

$$\dfrac{\dfrac{\dfrac{}{C \cup R\,C_0, A \vdash x : \forall(\vec{\alpha}\vec{\beta} : C_0).\, t_0 \,\&\, \emptyset}\,(\mathrm{id})}{C \cup R\,C_0, A \vdash x : R\,t_0 \,\&\, \emptyset}\,(\mathrm{ins})}{C, A \vdash x : \forall(\vec{\alpha'}\vec{\beta'} : R\,C_0).\, R\,t_0 \,\&\, \emptyset}\,(\mathrm{gen})$$

The rule (ins) is applicable since $Dom(R) \subseteq \{\vec{\alpha}\vec{\beta}\}$ and $C \cup R\,C_0 \vdash R\,C_0$. The rule (gen) is applicable because $\forall(\vec{\alpha}\vec{\beta} : C_0).\, t_0 = \forall(\vec{\alpha'}\vec{\beta'} : R\,C_0).\, R\,t_0$ (up to alpha-renaming) by assumption is well-formed and solvable from C, and furthermore $\{\vec{\alpha'}\vec{\beta'}\} \cap FV(C, A, \emptyset) = \emptyset$ holds by choice of $\vec{\alpha'}\vec{\beta'}$.

The case (abs). Then we have $C, A \vdash_c \mathtt{fn}\ x \Rightarrow e : t_1 \to^b t_2 \,\&\, \emptyset$ because $C, A[x : t_1] \vdash_c e : t_2 \,\&\, b$. Since t_1 is well-formed and solvable from C we can apply the induction hypothesis and get $C, A[x : t_1] \vdash_n e : t_2 \,\&\, b$ from which we infer $C, A \vdash_n \mathtt{fn}\ x \Rightarrow e : t_1 \to^b t_2 \,\&\, \emptyset$.

The case (app). Then we have $C, A \vdash_c e_1\, e_2 : t_1 \,\&\, (b_1 \cup b_2 \cup b)$ because $C, A \vdash_c e_1 : t_2 \to^b t_1 \,\&\, b_1$ and $C, A \vdash_c e_2 : t_2 \,\&\, b_2$. Then the induction hypothesis gives $C, A \vdash_n e_1 : t_2 \to^b t_1 \,\&\, b_1$ and $C, A \vdash_n e_2 : t_2 \,\&\, b_2$. We thus can infer the desired $C, A \vdash_n e_1\, e_2 : t_1 \,\&\, (b_1 \cup b_2 \cup b)$.

The case (let). Then we have $C, A \vdash_c \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : t_2 \,\&\, (b_1 \cup b_2)$ because $C, A \vdash_c e_1 : ts_1 \,\&\, b_1$ and $C, A[x : ts_1] \vdash_c e_2 : t_2 \,\&\, b_2$. Then the induction hypothesis gives $C, A \vdash_n e_1 : ts_1 \,\&\, b_1$. From Fact 2.14 we get that ts_1

is well-formed and solvable from C, so we can apply the induction hypothesis to get $C, A[x : ts_1] \vdash_n e_2 : t_2 \,\&\, b_2$. This enables us to infer the desired $C, A \vdash_n \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t_2 \,\&\, (b_1 \cup b_2)$.

The cases (rec), (if), (sub): Analogous to the above cases.

The case (ins). Then $C, A \vdash_c e : S\,t_0 \,\&\, b$ because $C, A \vdash_c e : \forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0 \,\&\, b$ where $Dom(S) \subseteq \{\vec{\alpha}\vec{\beta}\}$ and $C \vdash S\,C_0$. By applying the induction hypothesis we get $C, A \vdash_n e : \forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0 \,\&\, b$ where this inference tree has the form

$$\dfrac{\dfrac{\vdots}{C \cup C_0, A \vdash_n e : t_0 \,\&\, b}}{C, A \vdash_n e : \forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0 \,\&\, b} \quad \text{(gen)}$$

Since (gen) is applied we know that $\{\vec{\alpha}\vec{\beta}\} \cap FV(C, A, b) = \emptyset$. From Lemma 2.15 we therefore get

$$C \cup S\,C_0, A \vdash_n e : S\,t_0 \,\&\, b$$

and using Lemma 2.16 we get $C, A \vdash_n e : S\,t_0 \,\&\, b$ as desired.

The case (gen). Then we have $C, A \vdash_c e : \forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0 \,\&\, b$ because $C \cup C_0, A \vdash_c e : t_0 \,\&\, b$ where $\forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0$ is well-formed, solvable from C and satisfies $\{\vec{\alpha}\vec{\beta}\} \cap FV(C, A, b) = \emptyset$. Now A is well-formed and solvable from $C \cup C_0$ so the induction hypothesis gives $C \cup C_0, A \vdash_n e : t_0 \,\&\, b$. Therefore we have the TS-normalised inference tree $C, A \vdash_n e : \forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0 \,\&\, b$. □

Conservative extension

Here we shall prove Theorem 2.27, but first we must develop the necessary machinery.

First some auxiliary notions: we say that a constraint set C is sequential if all constraints in C are of form $\beta_1 \subseteq \beta_2$; and the set of free *type variables* in some entity g is denoted $FTV(g)$.

Next we introduce the notion of simplicity: a type is *simple* if all its behaviour annotations are behaviour variables; a sequential type scheme is simple if its type is; an assumption list is simple if all its type schemes are; finally a substitution is simple if it maps behaviour variables to behaviour variables and type variables to simple types.

A type or type scheme is said to be *essentially simple* if it is simple except that some arrows in covariant position are annotated with $\emptyset$, because these annotations can be replaced by fresh (bound) behaviour variables without changing the set of "instances" (the result of first applying (ins) and then applying (sub)).

Fact A.2 For all sequential constants c, the type scheme $\mathrm{TypeOf}(c)$ is essentially simple.

Fact A.3 For all simple or essentially simple types t, it holds that $FV(\epsilon(t)) \subseteq FV(t)$ and that $FTV(\epsilon(t)) = FTV(t)$.

For all simple and sequential type schemes ts, it holds that $FV(\epsilon(ts)) \subseteq FV(ts)$ and that $FTV(\epsilon(ts)) = FTV(ts)$. $\square$

From a substitution S we construct an ML substitution $R = \epsilon(S)$ as follows: $R\,\alpha = \epsilon(S\,\alpha)$.

Fact A.4 For all substitutions S and types t, we have $\epsilon(S\,t) = \epsilon(S)\,\epsilon(t)$.

Proof Induction in t. If $t = \alpha$, the equation follows from the definition of $\epsilon(S)$. If t is a base type like `int`, the equation is trivial. If t is a composite type like $t_1 \to^b\ t_2$, the equation reads

$$\epsilon(S\,t_1) \to\ \epsilon(S\,t_2) = \epsilon(S)\,\epsilon(t_1) \to\ \epsilon(S)\,\epsilon(t_2)$$

and follows from the induction hypothesis. If t is a non-sequential type like $t'\ \mathtt{com}\ b$, the equation reads $\epsilon(S\,t') = \epsilon(S)\,\epsilon(t')$ which follows from the induction hypothesis. $\square$

Proof of the first part of Theorem 2.27

The first part of the theorem follows from the following proposition, which admits a proof by induction, showing that there exists β and sequential C and sequential t with $\epsilon(t) = u$ such that $C, \emptyset \vdash e : t \,\&\, \beta$. Now let S be a substitution which maps all behaviour variables into $\emptyset$ and which leaves all type variables unchanged; then apply Lemma 2.15 and Lemma 2.16 to get $\emptyset, \emptyset \vdash e : S\,t \,\&\, \emptyset$ where clearly $S\,t$ is sequential with $\epsilon(S\,t) = \epsilon(t) = u$.

Proposition A.5 Let e be sequential. Suppose $A \vdash_{\mathrm{ML}} e : us$ and that A' is simple and sequential with $\epsilon(A') = A$. Then there exists sequential C, simple and sequential ts with $\epsilon(ts) = us$, and β such that $C, A' \vdash e : ts \,\&\, \beta$. Similarly with u and t instead of us and ts.

We need the following auxiliary result:

Fact A.6 Suppose t and t' are simple and sequential and that $\epsilon(t) = \epsilon(t')$. Then there exists sequential C such that $C \vdash t \equiv t'$.

Proof Induction in t: if $t = \alpha$ then $\epsilon(t') = \alpha$ so from t' being sequential we deduce that $t' = \alpha$, hence the claim (with $C = \emptyset$).

Now consider the case where t is a composite type like $t_1 \to^b t_2$. Then $\epsilon(t') = \epsilon(t_1) \to \epsilon(t_2)$ so from t' being sequential we deduce that t' is of form $t_1' \to^{b'} t_2'$, with $\epsilon(t_1') = \epsilon(t_1)$ and $\epsilon(t_2') = \epsilon(t_2)$. The induction hypothesis then tells us that there exists sequential C_1, C_2 such that $C_1 \vdash t_1 \equiv t_1'$ and $C_2 \vdash t_2 \equiv t_2'$. As t and t' are simple it holds that b and b' are both variables; therefore the constraint set $C = C_1 \cup C_2 \cup \{b \subseteq b',\ b' \subseteq b\}$ is sequential and clearly $C \vdash t \equiv t'$. $\square$

We now embark on proving Proposition A.5 by induction in the proof tree for $A \vdash_{\mathrm{ML}} e : us$, where we perform case analysis on the definition in Fig. 4 (where the clauses for conditionals and for recursion are omitted, as they present no further complications).

The case (con): By Assumption 2.26 (and Fact 2.25) together with Fact A.2 we can use $ts = \mathrm{TypeOf}(c)$ and $C = \emptyset$; in order to get from $\emptyset, A' \vdash c : ts \,\&\, \emptyset$ to $\emptyset, A' \vdash c : ts \,\&\, \beta$ (with β a fresh variable) we can use (sub) since $\emptyset \vdash \emptyset \subseteq \beta$.

The case (id): Trivial; as in the previous case we use (sub).

The case (abs): We can clearly find simple and sequential t_1 such that $\epsilon(t_1) = u_1$. Then $\epsilon(A'[x : t_1]) = A[x : u_1]$, so we can apply the induction hypothesis to infer that there exists sequential C, simple and sequential t_2 with $\epsilon(t_2) = u_2$ and β such that

$$C, A'[x : t_1] \vdash e : t_2 \,\&\, \beta.$$

Let β' be a fresh variable, then by using (abs) and (sub) we are able to infer

$$C, A' \vdash \texttt{fn}\ x \Rightarrow e : t_1 \to^{\beta} t_2 \,\&\, \beta'$$

where the conclusion is as desired since $\epsilon(t_1 \to^{\beta} t_2) = u_1 \to u_2$.

The case (app): We can apply the induction hypothesis to find sequential C_1 and C_2, behaviour variables β_1 and β_2, and simple and sequential t_1' and t_2' with $\epsilon(t_1') = u_2 \to u_1$ and $\epsilon(t_2') = u_2$, such that

$$C_1, A' \vdash e_1 : t_1' \,\&\, \beta_1 \text{ and } C_2, A' \vdash e_2 : t_2' \,\&\, \beta_2.$$

Clearly there exists β and simple and sequential t_2, t_1 such that $t_1' = t_2 \to^{\beta} t_1$, and $\epsilon(t_2) = u_2$ and $\epsilon(t_1) = u_1$. By Fact A.6 there exists sequential C' such that $C' \vdash t_2' \equiv t_2$. Hence by (sub) we have

$$C_1, A' \vdash e_1 : t_2 \to^{\beta} t_1 \,\&\, \beta_1 \text{ and } C_2 \cup C', A' \vdash e_2 : t_2 \,\&\, \beta_2$$

so by (app) we are able to infer

$$C_1 \cup C_2 \cup C', A' \vdash e_1\, e_2 : t_1 \,\&\, \beta_1 \cup \beta_2 \cup \beta.$$

Let $C = C_1 \cup C_2 \cup C' \cup \{\beta_1 \subseteq \beta, \beta_2 \subseteq \beta\}$, then by (sub) we have

$$C, A' \vdash e_1\, e_2 : t_1 \,\&\, \beta$$

which is as desired since $\epsilon(t_1) = u_1$ and since C is sequential.

The case (let): We can apply the induction hypothesis to find sequential C_1, simple and sequential ts_1 with $\epsilon(ts_1) = us_1$ and β_1 such that

$$C_1, A' \vdash e_1 : ts_1 \,\&\, \beta_1.$$

Since $\epsilon(A'[x : ts_1]) = A[x : us_1]$ we can apply the induction hypothesis to find sequential C_2, simple and sequential t_2 with $\epsilon(t_2) = u_2$ and β_2 such that

$$C_2, A'[x : ts_1] \vdash e_2 : t_2 \,\&\, \beta_2$$

Let $C = C_1 \cup C_2 \cup \{\beta_2 \subseteq \beta_1\}$, then we can apply (let) and (sub) to get the desired judgement

$$C, A' \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : t_2 \,\&\, \beta_1.$$

The case (ins): We can apply the induction hypothesis to find β, sequential C and simple and sequential ts with $\epsilon(ts) = \forall \vec{\alpha}.u$ such that

$$C, A' \vdash e : ts \,\&\, \beta.$$

Here ts is of form $\forall(\vec{\alpha}\,\vec{\beta} : \emptyset).\ t_0$ where $u = \epsilon(t_0)$ with t_0 simple and sequential. It is clearly possible to find a simple substitution S with $Dom(S) \subseteq \{\vec{\alpha}\}$ such that $\epsilon(S) = R$ and such that $S\, t_0$ is sequential and simple. But then (ins) gives us the judgement

$$C, A' \vdash e : S\, t_0 \,\&\, \beta$$

which is as desired since by Fact A.4 we have $\epsilon(S\, t_0) = R\, u$.

The case (gen): We can apply the induction hypothesis to find β, sequential C and simple and sequential t with $\epsilon(t) = u$ such that

$$C, A' \vdash e \,:\, t \,\&\, \beta$$

and the conclusion we want to arrive at is

$$C, A' \vdash e \,:\, \forall(\vec{\alpha} : \emptyset).\ t \,\&\, \beta$$

which follows by using (gen) provided that (i) $\forall(\vec{\alpha} : \emptyset).\ t$ is well-formed and solvable from C and (ii) $\vec{\alpha} \cap (FV(A') \cup FV(C) \cup \{\beta\}) = \emptyset$. Here (i) is trivial; and (ii) follows since we from Fact A.3 have $FTV(A) = FTV(A')$.

Auxiliary notions.

Before embarking on the second part of Theorem 2.27 we need to develop some extra machinery.

ML type equations. ML type equations are of the form $u_1 = u_2$. With C_t a set of ML type equations and with R an ML substitution, we say that R satisfies (or unifies) C_t iff for all $(u_1 = u_2) \in C_t$ we have $R\,u_1 = R\,u_2$.

The following fact is well-known from unification theory:

Fact A.7 Let C_t be a set of ML type equations. If there exists an ML substitution which satisfies C_t, then C_t has a "most general unifier": that is, an idempotent substitution R which satisfies C_t such that if R' also satisfies C_t then there exists R'' such that $R' = R''\,R$.

Lemma A.8 Suppose R_0 with $Dom(R_0) \subseteq G$ satisfies a set of ML type equations C_t. Then C_t has a most general unifier R with $Dom(R) \subseteq G$.

Proof From Fact A.7 we know that C_t has a most general unifier R_1, and hence there exists R_2 such that $R_0 = R_2\,R_1$. Let $G_1 = Dom(R_1) \setminus Dom(R_0)$; for $\alpha \in G_1$ we have $R_2\,R_1\,\alpha = R_0\,\alpha = \alpha$ and hence R_1 maps the variables in G_1 into distinct variables G_2 (which by R_2 are mapped back again). Since R_1 is idempotent we have $G_2 \cap Dom(R_1) = \emptyset$, so R_0 equals R_2 on G_2 showing that $G_2 \subseteq Dom(R_0)$. Moreover, $G_1 \cap G_2 = \emptyset$.

Let ϕ map $\alpha \in G_1$ into $R_1\,\alpha$ and map $\alpha \in G_2$ into $R_2\,\alpha$ and behave as the identity otherwise. Then ϕ is its own inverse so that $\phi\,\phi = \mathrm{Id}$. Now define $R = \phi\,R_1$; clearly R unifies C_t and if R' also unifies C_t then (since R_1 is most general unifier) there exists R'' such that $R' = R''\,R_1 = R''\,\phi\,\phi\,R_1 = (R''\,\phi)\,R$.

We are left with showing (i) that R is idempotent and (ii) that $Dom(R) \subseteq G$. For (i), first observe that $R_1\,\phi$ equals Id except on $Dom(R_1)$. Since R_1 is idempotent we have $FV(R_1\,\alpha)\,\cap\,Dom(R_1) = \emptyset$ (for all α) and hence

$$R\,R = \phi\,R_1\,\phi\,R_1 = \phi\,\mathrm{Id}\,R_1 = R.$$

For (ii), observe that R equals Id on G_1 so it will be sufficient to show that $R\,\alpha = \alpha$ if $\alpha \notin (G \cup G_1)$. But then $\alpha \notin Dom(R_0)$ and hence $\alpha \notin G_2$ and $\alpha \notin Dom(R_1)$ so $R\,\alpha = \phi\,\alpha = \alpha$. □

From a constraint set C we construct a set of ML type equations $\epsilon(C)$ as follows:

$$\epsilon(C) = \{(\epsilon(t_1) = \epsilon(t_2)) \mid (t_1 \subseteq t_2)\ \in\ C\}.$$

Fact A.9 Suppose $C \vdash t_1 \subseteq t_2$. If R satisfies $\epsilon(C)$ then $R\,\epsilon(t_1) = R\,\epsilon(t_2)$.

So if $C \vdash C'$ and R satisfies $\epsilon(C)$ then R satisfies $\epsilon(C')$.

Proof Induction in the proof tree. If $(t_1 \subseteq t_2)\ \in\ C$, the claim follows from the assumptions. The cases for reflexivity and transitivity are straight-forward. For the structural rules with the "sequential" type constructors, assume e.g. that $C \vdash t_1 \to^{b}\ t_2 \subseteq t_1' \to^{b'}\ t_2'$ because (among other things) $C \vdash t_1' \subseteq t_1$ and $C \vdash t_2 \subseteq t_2'$. By using the induction hypothesis we get the desired equality

$$R\,\epsilon(t_1 \to^{b}\ t_2) = R\,\epsilon(t_1) \to\ R\,\epsilon(t_2) = R\,\epsilon(t_1') \to\ R\,\epsilon(t_2') = R\,\epsilon(t_1' \to^{b'}\ t_2').$$

For the structural rules with the non-sequential type constructors, assume e.g. that $C \vdash t\ \mathtt{com}\ b \subseteq t'\ \mathtt{com}\ b'$ where $C \vdash t \subseteq t'$. Then the desired equality reads $R\,\epsilon(t) = R\,\epsilon(t')$ and follows from the induction hypothesis. □

Relating type schemes. For a type scheme $ts = \forall(\vec{\alpha}\,\vec{\beta} : C).\ t$ we shall not in general (when $C\ \neq\ \emptyset$) define any entity $\epsilon(ts)$; this is because one natural attempt, namely $\forall(\vec{\alpha} : \epsilon(C)).\ \epsilon(t)$, is not an ML type scheme and another natural attempt, $\forall\vec{\alpha}.\epsilon(t)$, causes loss of the information in $\epsilon(C)$. Rather we shall define some relations between ML types, types, ML type schemes and type schemes:

Definition A.10 We write $u \prec_{\epsilon}^{R} ts$, where $ts = \forall(\vec{\alpha}\,\vec{\beta} : C_0).\ t_0$ and where R is an ML substitution, iff there exists R_0 which equals R on all variables except $\vec{\alpha}$ such that R_0 satisfies $\epsilon(C_0)$ and such that $u = R_0\,\epsilon(t_0)$.

Notice that instead of demanding R_0 to equal R on all variables but $\vec{\alpha}$, it is sufficient to demand that R_0 equals R on $FTV(ts)$. Hence we have the expected property that if $u \prec_{\epsilon}^{R} ts$ and ts is alpha-equivalent to ts' then also $u \prec_{\epsilon}^{R} ts'$.

Definition A.11 We write $u \prec us$, where $us = \forall \vec{\alpha}.u_0$, iff there exists R_0 with $Dom(R_0) \subseteq \vec{\alpha}$ such that $u = R_0\, u_0$.

Definition A.12 We write $us \cong^R_\epsilon ts$ to mean that (for all u) $u \prec us$ iff $u \prec^R_\epsilon ts$.

Fact A.13 Suppose $us = \epsilon(ts)$, where $ts = \forall(\vec{\alpha}\vec{\beta} : \emptyset).\ t$ is sequential. Then $us \cong^{\mathrm{Id}}_\epsilon ts$.

Proof We have $us = \forall \vec{\alpha}.\epsilon(t)$, so for any u it holds that $u \prec us \Leftrightarrow \exists\ R$ with $Dom(R) \subseteq \vec{\alpha}$ such that $u = R\,\epsilon(t) \Leftrightarrow u \prec^{\mathrm{Id}}_\epsilon ts$. □

Notice that $\forall().u \cong^R_\epsilon \forall(() : \emptyset).\ t_0$ holds iff $u = R\,\epsilon(t_0)$. We can thus consistently extend $\cong^R_\epsilon$ to relate not only type schemes but also types:

Definition A.14 We write $u \cong^R_\epsilon t$ iff $u = R\,\epsilon(t)$.

Definition A.15 We write $A' \cong^R_\epsilon A$ iff $Dom(A') = Dom(A)$ and $A'(x) \cong^R_\epsilon A(x)$ for all $x \in Dom(A)$.

Fact A.16 Let R and S be such that $\epsilon(S) = R$. Then the relation $u \prec^R_\epsilon ts$ holds iff the relation $u \prec^{\mathrm{Id}}_\epsilon S\,ts$ holds.

Consequently, $us \cong^R_\epsilon ts$ holds iff $us \cong^{\mathrm{Id}}_\epsilon S\,ts$ holds.

Proof Let $ts = \forall(\vec{\alpha}\vec{\beta} : C).\ t$. Due to the remark after Definition A.10 we can assume that $\vec{\alpha}\vec{\beta}$ is disjoint from $Dom(S) \cup Ran(S)$, so $S\,ts = \forall(\vec{\alpha}\vec{\beta} : S\,C).\ S\,t$.

First we prove "if". For this suppose that R' equals Id except on $\vec{\alpha}$ and that R' satisfies $\epsilon(S\,C)$ and that $u = R'\,\epsilon(S\,t)$, which by straight-forward extensions of Fact A.4 amounts to saying that R' satisfies $R\,\epsilon(C)$ and that $u = R'\,R\,\epsilon(t)$. Since $\{\vec{\alpha}\} \cap Ran(R) = \emptyset$ we conclude that $R'\,R$ equals R except on $\vec{\alpha}$, so we can use $R'\,R$ to show that $u \prec^R_\epsilon ts$.

Next we prove "only if". For this suppose that R' equals R except on $\vec{\alpha}$ and that R' satisfies $\epsilon(C)$ and that $u = R'\,\epsilon(t)$. Let R'' behave as R' on $\vec{\alpha}$ and behave as the identity otherwise. Our task is to show that R'' satisfies $\epsilon(S\,C)$ and that $u = R''\,\epsilon(S\,t)$, which as we saw above amounts to showing that R'' satisfies $R\,\epsilon(C)$ and that $u = R''\,R\,\epsilon(t)$. This will follow if we can show that $R' = R''\,R$. But if $\alpha \in \vec{\alpha}$ we have $R''\,R\,\alpha = R''\,\alpha = R'\,\alpha$ since $Dom(R) \cap \{\vec{\alpha}\} = \emptyset$, and if $\alpha \notin \vec{\alpha}$ we have $R''\,R\,\alpha = R\,\alpha = R'\,\alpha$ where the first equality sign follows from $Ran(R) \cap \{\vec{\alpha}\} = \emptyset$ and $Dom(R'') \subseteq \vec{\alpha}$. □

Fact A.17 If $us \cong^{\mathrm{Id}}_\epsilon ts$ then $FV(us) \subseteq FV(ts)$.

Proof We assume $us \cong^{\mathrm{Id}}_{\epsilon} ts$ where $us = \forall \vec{\alpha}\,'.u$ and $ts = \forall(\vec{\alpha}\vec{\beta} : C).\ t$. Let α_1 be given such that $\alpha_1 \notin FV(ts)$, our task is to show that $\alpha_1 \notin FV(us)$.

Clearly $u \prec us$ so $u \prec^{\mathrm{Id}}_{\epsilon} ts$, that is there exists R with $Dom(R) \subseteq \vec{\alpha}$ such that R satisfies $\epsilon(C)$ and such that $u = R\,\epsilon(t)$. Now define a substitution R_1 which maps α_1 into a fresh variable and is the identity otherwise. Due to our assumption about α_1 it is easy to see that $R_1\,R$ equals Id on $FV(ts)$, and as $R_1\,R$ clearly satisfies $\epsilon(C)$ it holds that $R_1\,u = R_1\,R\,\epsilon(t) \prec^{\mathrm{Id}}_{\epsilon} ts$ and hence also $R_1\,u \prec us$. As $\alpha_1 \notin FV(R_1\,u)$ we can infer the desired $\alpha_1 \notin FV(us)$. □

Proof of the second part of Theorem 2.27

The second part of the theorem follows from the following proposition which admits a proof by induction.

Proposition A.18 Let e be sequential, suppose $C, A \vdash e : ts \,\&\, b$, suppose R satisfies $\epsilon(C)$, and suppose $A' \cong^{R}_{\epsilon} A$; then there exists a us with $us \cong^{R}_{\epsilon} ts$ such that $A' \vdash_{\mathrm{ML}} e : us$. Similarly with t and u instead of ts and us (in which case $u = R\,\epsilon(t)$).

We perform induction in the proof tree (the clauses for conditionals and for recursion are omitted, as they present no further complications):

The case (con): Suppose R satisfies $\epsilon(C)$, and suppose $A' \cong^{R}_{\epsilon} A$. We can infer $A' \vdash_{\mathrm{ML}} c : \mathrm{MLTypeOf}(c)$ so we must show $\mathrm{MLTypeOf}(c) \cong^{R}_{\epsilon} \mathrm{TypeOf}(c)$.

By Assumption 2.26 and by Fact A.13 we know that $\mathrm{MLTypeOf}(c) \cong^{\mathrm{Id}}_{\epsilon} \mathrm{TypeOf}(c)$. There clearly exists S with $\epsilon(S) = R$, so the claim follows from Fact A.16, since $\mathrm{TypeOf}(c)$ is closed (cf. Fact 2.13).

The case (id): Suppose R satisfies $\epsilon(C)$, and suppose $A' \cong^{R}_{\epsilon} A$. Then $A'(x) \cong^{R}_{\epsilon} A(x)$ and $A' \vdash_{\mathrm{ML}} x : A'(x)$, as desired.

The case (abs): Suppose R satisfies $\epsilon(C)$ and that $A' \cong^{R}_{\epsilon} A$. Then also $A'[x : R\,\epsilon(t_1)] \cong^{R}_{\epsilon} A[x : t_1]$, so the induction hypothesis can be applied to find u_2 such that $u_2 = R\,\epsilon(t_2)$ and such that $A'[x : R\,\epsilon(t_1)] \vdash_{\mathrm{ML}} e : u_2$. By using (abs) we get the judgement

$$A' \vdash_{\mathrm{ML}} \texttt{fn}\ x \Rightarrow e : R\,\epsilon(t_1) \rightarrow\ u_2$$

which is as desired since $R\,\epsilon(t_1) \rightarrow\ u_2 = R\,\epsilon(t_1 \rightarrow^{b}\ t_2)$.

The case (app): Suppose R satisfies $\epsilon(C_1 \cup C_2)$ and that $A' \cong^R_\epsilon A$. Clearly R satisfies $\epsilon(C_1)$ as well as $\epsilon(C_2)$, so the induction hypothesis can be applied to infer that

$$A' \vdash_{\mathrm{ML}} e_1 : R\,\epsilon(t_2 \to^b t_1) \text{ and } A' \vdash_{\mathrm{ML}} e_2 : R\,\epsilon(t_2)$$

and since $R\,\epsilon(t_2 \to^b t_1) = R\,\epsilon(t_2) \to R\,\epsilon(t_1)$ we can apply (app) to arrive at the desired judgement $A' \vdash_{\mathrm{ML}} e_1\, e_2 : R\,\epsilon(t_1)$.

The case (let): Suppose R satisfies $\epsilon(C_1 \cup C_2)$ and that $A' \cong^R_\epsilon A$. Since R satisfies $\epsilon(C_1)$ we can apply the induction hypothesis to find us_1 such that $us_1 \cong^R_\epsilon ts_1$ and such that $A' \vdash_{\mathrm{ML}} e_1 : us_1$.

Since R satisfies $\epsilon(C_2)$ and since $A'[x : us_1] \cong^R_\epsilon A[x : ts_1]$ we can apply the induction hypothesis to infer that $A'[x : us_1] \vdash_{\mathrm{ML}} e_2 : R\,\epsilon(t_2)$. Now use (let) to arrive at the desired judgement $A' \vdash_{\mathrm{ML}} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : R\,\epsilon(t_2)$.

The case (sub): Suppose R satisfies $\epsilon(C)$ and that $A' \cong^R_\epsilon A$. By applying the induction hypothesis we infer that $A' \vdash_{\mathrm{ML}} e : R\,\epsilon(t)$ and since by Fact A.9 we have $R\,\epsilon(t) = R\,\epsilon(t')$ this is as desired.

The case (ins): Suppose that R satisfies $\epsilon(C)$ and that $A' \cong^R_\epsilon A$. The induction hypothesis tells us that there exists us with $us \cong^R_\epsilon \forall(\vec{\alpha}\,\vec{\beta} : C_0).\ t_0$ such that $A' \vdash_{\mathrm{ML}} e : us$.

Since $C \vdash S_0\, C_0$ and R satisfies $\epsilon(C)$, Fact A.9 tells us that R satisfies $\epsilon(S_0\, C_0)$ which by Fact A.4 equals $\epsilon(S_0)\,\epsilon(C_0)$, thus $R\,\epsilon(S_0)$ satisfies $\epsilon(C_0)$. As $R\,\epsilon(S_0)$ equals R except on $\vec{\alpha}$, it holds that $R\,\epsilon(S_0)\,\epsilon(t_0) \prec^R_\epsilon \forall(\vec{\alpha}\,\vec{\beta} : C_0).\ t_0$ and since $us \cong^R_\epsilon \forall(\vec{\alpha}\,\vec{\beta} : C_0).\ t_0$ we have $R\,\epsilon(S_0)\,\epsilon(t_0) \prec us$. But this shows that we can use (ins) to arrive at the judgement $A' \vdash_{\mathrm{ML}} e : R\,\epsilon(S_0)\,\epsilon(t_0)$ which is as desired since $\epsilon(S_0)\,\epsilon(t_0) = \epsilon(S_0\, t_0)$ by Fact A.4.

The case (gen): Suppose that R satisfies $\epsilon(C)$ and that $A' \cong^R_\epsilon A$. Our task is to find us such that $us \cong^R_\epsilon \forall(\vec{\alpha}\,\vec{\beta} : C_0).\ t_0$ and such that $A' \vdash_{\mathrm{ML}} e : us$. Below we will argue that we can assume that $\{\vec{\alpha}\} \cap (Dom(R) \cup Ran(R)) = \emptyset$.

> Let T be a renaming substitution mapping $\vec{\alpha}$ into fresh variables $\vec{\alpha}\,'$. By applying Lemma 2.15, by exploiting that $FV(C, A, b) \cap \{\vec{\alpha}\,\vec{\beta}\} = \emptyset$, and by using (gen) we can construct a proof tree whose last nodes are
>
> $$\frac{C \cup T\,C_0, A \vdash e : T\,t_0 \,\&\, b}{C, A \vdash e : \forall(\vec{\alpha}\,'\vec{\beta} : T\,C_0).\ T\,t_0 \,\&\, b}$$

the conclusion of which is alpha-equivalent to the conclusion of the original proof tree, and the shape of which (by Lemma 2.15) is equal to the shape of the original proof tree.

There exists S_0 with $Dom(S_0) \subseteq \{\vec{\alpha}\,\vec{\beta}\}$ such that $C \vdash S_0\, C_0$. Fact A.9 then tells us that R satisfies $\epsilon(S_0\, C_0)$ which by Fact A.4 equals $\epsilon(S_0)\, \epsilon(C_0)$.

Now define R_0' to be a substitution with $Dom(R_0') \subseteq \{\vec{\alpha}\}$ which maps $\vec{\alpha}$ into $R\, \epsilon(S_0)\, \vec{\alpha}$. It is easy to see (since $\vec{\alpha}$ is disjoint from $Dom(R) \cup Ran(R)$) that $R_0'\, R = R\, \epsilon(S_0)$, implying that R_0' satisfies $R\, \epsilon(C_0)$.

By Lemma A.8 there exists R_0 with $Dom(R_0) \subseteq \{\vec{\alpha}\}$ which is a most general unifier of $R\, \epsilon(C_0)$. Hence with $R' = R_0\, R$ it holds not only that R' satisfies $\epsilon(C)$ but also that R' satisfies $\epsilon(C_0)$, so in order to apply the induction hypothesis on R' we just need to show that $A' \cong_\epsilon^{R'} A$. This can be done by showing that R equals R' on $FV(A)$, but this follows since our assumptions tell us that $Dom(R_0) \cap FV(R\, A) = \emptyset$.

The induction hypothesis thus tells us that $A' \vdash_{\mathrm{ML}} e : R'\, \epsilon(t_0)$. Let S be such that $\epsilon(S) = R$ and $Dom(S) = Dom(R)$ and $Ran(S) \cap \{\vec{\beta}\} = \emptyset$; since $\{\vec{\alpha}\} \cap Ran(R) = \emptyset$ we can also obtain $\{\vec{\alpha}\} \cap Ran(S) = \emptyset$. By Fact A.16 and Fact A.17 we infer that $FV(A') \subseteq FV(S\, A)$, so since $\{\vec{\alpha}\} \cap FV(A) = \emptyset$ we infer $\{\vec{\alpha}\} \cap FV(A') = \emptyset$. We can thus use (gen) to arrive at the judgement $A' \vdash_{\mathrm{ML}} e : \forall \vec{\alpha}\, .R'\, \epsilon(t_0)$.

We are left with showing that $\forall \vec{\alpha}\, .R'\, \epsilon(t_0) \cong_\epsilon^R \forall(\vec{\alpha}\,\vec{\beta} : C_0).\ t_0$ but this follows from the following calculation (explained below):

$$
\begin{array}{ll}
& u \prec_\epsilon^R \forall(\vec{\alpha}\,\vec{\beta} : C_0).\ t_0 \\
\Leftrightarrow & u \prec_\epsilon^{\mathrm{Id}} \forall(\vec{\alpha}\,\vec{\beta} : S\, C_0).\ S\, t_0 \\
\Leftrightarrow & \exists R_1 \text{ with } Dom(R_1) \subseteq \{\vec{\alpha}\} \\
& \qquad \text{such that } R_1 \text{ satisfies } R\, \epsilon(C_0) \text{ and } u = R_1\, R\, \epsilon(t_0) \\
\Leftrightarrow & \exists R_1 \text{ with } Dom(R_1) \subseteq \{\vec{\alpha}\} \\
& \qquad \text{such that } \exists R_2 : R_1 = R_2\, R_0 \text{ and } u = R_1\, R\, \epsilon(t_0) \\
\Leftrightarrow & \exists R_2 \text{ with } Dom(R_2) \subseteq \{\vec{\alpha}\} \text{ such that } u = R_2\, R_0\, R\, \epsilon(t_0) \\
\Leftrightarrow & u \prec \forall \vec{\alpha}\, .R'\, \epsilon(t_0).
\end{array}
$$

The first $\Leftrightarrow$ follows from Fact A.16 where we have exploited that $\{\vec{\alpha}\,\vec{\beta}\}$ is disjoint from $Dom(S) \cup Ran(S)$; the second $\Leftrightarrow$ follows from the definition of $\prec_\epsilon^{\mathrm{Id}}$ together with Fact A.4; the third $\Leftrightarrow$ is a consequence of R_0 being the most general unifier of $R\, \epsilon(C_0)$; and the fourth $\Leftrightarrow$ is a consequence of $Dom(R_0) \subseteq \{\vec{\alpha}\}$ since then from $R_1 = R_2\, R_0$ we conclude that if $\alpha' \notin \{\vec{\alpha}\}$ then $R_1\, \alpha' = R_2\, \alpha'$ and hence $Dom(R_1) \subseteq \{\alpha\}$ iff $Dom(R_2) \subseteq \{\alpha\}$.