# Polymorphic Subtyping for Effect Analysis: the Static Semantics

Hanne Riis Nielson & Flemming Nielson & Torben Amtoft

Computer Science Department, Aarhus University, Denmark

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

**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 [10] that developed the first polymorphic type inference and algorithm for the applicative fragment of ML; a shorter presentation for the typed $\lambda$-calculus with `let` is given in [3].

Since then many papers have studied how to integrate subtyping. A number of early papers did so by mainly focusing on the typed $\lambda$-calculus and only briefly dealing with `let` [11, 5]. Later papers have treated polymorphism in full generality [18, 8]. A key ingredient in these approaches is the simplification of the enormous set of constraints into something manageable [4, 18].

Already ML necessitates an incorporation of imperative effects due to the presence of `ref`-types. A pioneering paper in the area is [21] that develops a distinction between imperative and applicative type variables: for *creation* of a reference cell we demand that its type contain imperative variables only; and one is not allowed to generalise over imperative variables unless the expression in question is *non-expansive* (i.e. does not expand the store) which will be the case if it is an identifier or a function abstraction.

The problem of typing ML with references (but without subtyping) has lead to a number of attempts to improve upon [21]; this includes the following:

- [23] is similar in spirit to [21] in that one is not allowed to generalise over a type variable if a reference cell has been *created* with a type containing this variable; to trace such variables the type system is augmented with *effects*. Effects may be approximated by larger effects, that is the system employs *subeffecting*.
- [19] can be considered a refinement of [23] in that effects also record the *region* in which a reference cell is created (or a read/write operation performed); this information enables one to "mask" effects which have taken place in "inaccessible" regions.
- [9] presents a somewhat alternative view: here focus is not on detecting *creation* of reference cells but rather to detect their *use*; this means that if an identifier occurs free in a function closure then all variables in its type have to be "examined". This method is quite powerful but unfortunately it fails to be a conservative extension of ML (cf. Sect. 2.6): some purely applicative programs which are typeable in ML may be untypeable in this system.

The surveys in [19, section 11] and in [23, section 5] show that many of these systems are incomparable, in the sense that for any two approaches it will often be the case that there are programs which are accepted by one of them but not by the other, and vice versa. Our approach (which will be illustrated by a fragment of Concurrent ML but is equally applicable to Standard ML with references) involves subtyping which is strictly more powerful than subeffecting (as shown in Example 4); apart from this we do not attempt to measure its strength relative to other approaches.

In the area of static program analysis, annotated type and effect systems have been used as the basis for control flow analysis [20] and binding time analysis [14, 7]. 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 [15] to let annotated type and effect systems extract terms of a process algebra from programs with

communication; this involves polymorphism and subeffecting but the algorithmic issues are non-trivial [12] (presumably because the inference system is expressed without using constraints); [1] presents an algorithm that is sound as well as complete, but which generates constraints that are not guaranteed to have best solutions. Finally we should mention [22] where effects are incorporated into ML types in order to deal with *region inference*.

**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 [19] and in admitting effects into the subtyping approaches of [18, 8]. The development is not only applicable to Concurrent ML (with communication) but also Standard ML (with references) and similar settings.

**The essence of Concurrent ML.** In this paper we study a fragment of Concurrent ML (CML) that as ML includes the $\lambda$-calculus and `let`-polymorphism, but which also includes five primitives for synchronous communication:

The constant `channel` allocates a new communication channel when applied to `()`.

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

The constant `sync` activates and synchronises a delayed communication created by either `send` or `receive`. Thus one process can send the value of $e$ to another process by the expression `sync(send(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 following program illustrates some of the essential features: it 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.

```
rec map2 f =>
   fn xs =>
     if isnil(xs) then []
     else let ch = channel ()
          in fork (fn d =>
                     (sync (send (ch, map2 f (tl xs)))));
             cons (f (hd xs))
                  (sync (receive ch))
```

That is, when applied to some function `f` and say the list `[x1,x2,x3]` the initial process $p_0$ will wait for a new process $p_1$ to send the value of `[f x2,f x3]` over a new channel $ch_1$ and in the meantime $p_0$ computes `f x1` such that it can finally return the value of `[f x1,f x2,f x3]`; $p_1$ will do its job by waiting for a new process $p_2$ to communicate the value of `[f x3]` over a new channel $ch_2$ and in the meantime $p_1$ computes `f x2`; etc.

**Overview.** We develop an annotated type and effect system in which a simple notion of behaviours is used to keep track of the types 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 [2, 13].

## 2 Inference System

The fragment of Concurrent ML [17, 16] we have chosen for illustrating our approach has *expressions* ($e \in Exp$) and *constants* ($c \in Con$) given by the following syntax:

$$\begin{aligned} e ::= {} & c \mid x \mid \texttt{fn}\ x \Rightarrow e \mid e_1\ e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \\ & \mid \texttt{rec}\ f\ x \Rightarrow e \mid \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \end{aligned}$$

$$\begin{aligned} c ::= {} & \texttt{()} \mid \texttt{true} \mid \texttt{false} \mid n \mid \texttt{+} \mid \texttt{*} \mid \texttt{=} \mid \cdots \\ & \mid \texttt{pair} \mid \texttt{fst} \mid \texttt{snd} \mid \texttt{nil} \mid \texttt{cons} \mid \texttt{hd} \mid \texttt{tl} \mid \texttt{isnil} \\ & \mid \texttt{send} \mid \texttt{receive} \mid \texttt{sync} \mid \texttt{channel} \mid \texttt{fork} \end{aligned}$$

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 ($n \in Num$), `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 $\texttt{(}e_1\texttt{,}e_2\texttt{)}$ for $\texttt{pair}\,e_1\,e_2$, to write `[]` for `nil` and $\texttt{[}e_1 \cdots e_n\texttt{]}$ for $\texttt{cons (}e_1\texttt{,cons(}\cdots\texttt{,nil)}\cdots\texttt{)}$. Additionally we shall write $e_1\texttt{;}e_2$ for $\texttt{snd (}e_1\texttt{,}e_2\texttt{)}$; to motivate this notice that since the language is call-by-value, evaluation of the latter expression will give rise to evaluation of $e_1$ followed by evaluation of $e_2$, the value of which will be the final result.

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 [17]. The non-sequential base functions are `channel` for allocating new communication channels, `fork` for spawning new processes, and `sync` for synchronising delayed communications; examples of their use are given in the Introduction.

**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 `ref` $e$ corresponds to `channel ()`, $e_1$`:=`$e_2$ corresponds to `sync (send (`$e_1$`,`$e_2$`))`, and `!`$e$ corresponds to `sync (receive` $e$`)`. Looking slightly ahead the Standard ML type $t$ `ref` will correspond to the Concurrent ML type $t$ `chan`. □

*Example 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 by a system using subeffecting only, whereas it will be accepted in the systems of [19] and [21]. 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:

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

Here we have base types for the unit type, booleans and integers; type variables are denoted $\alpha$; composite types include 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 [17]. 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 [19] and (a simplified version of) [15] the syntax of *effects*, or *behaviours*, ($b \in Beh$) is given by:

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

Apart from the presence of behaviour variables (denoted $\beta$) a behaviour can thus be viewed as a set of "atomic" behaviours each of form $\{t \text{ CHAN}\}$, recording the allocation of a channel of type $t$ `chan`. 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

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

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` (indirectly) 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 $\Rightarrow e$`)` may create new channels without this being recorded in the overall effect[2].

Following the approach of [18, 8] we will incorporate the effects of [19, 15] 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, $\sigma$ is a type $t$ or a type scheme $ts$, and $b$ is an

[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.

[2] To repair this one could add behaviours of form FORK $b$.

| $c$ | 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` | $\forall(\alpha\beta : \{\{\alpha\ \textsc{chan}\} \subseteq \beta\}).\ \texttt{unit} \to^{\beta} (\alpha\ \texttt{chan})$ |
| `fork` | $\forall(\alpha\beta : \emptyset).\ (\texttt{unit} \to^{\beta} \alpha) \to^{\emptyset} \texttt{unit}$ |

**Fig. 1.** Type schemes for selected constants.

effect. This means that $e$ has type or type scheme $\sigma$, 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 [18, 8] with a few components from [19, 15] thrown in; the novel ideas of our 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 (reflecting that the language is call-by-value). 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[3]. 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.

[3] Even though this is undecidable in general [6] one might allow polymorphic recursion in the annotations as in [7] or [22].

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

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

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

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

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

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

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

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

**Fig. 2.** The type inference system.

### 2.2 Subtyping

Rule (sub) generalises the subeffecting rule of [19] by incorporating subtyping and extends the subtyping rule of [18] 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 $C'$.

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 but there

**Ordering on behaviours**

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

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

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

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

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

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

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

**Ordering on types**

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

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

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

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

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

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

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

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

**Fig. 3.** Subtyping and subeffecting.

is no inclusion between base types. In the case of `chan` note that the type $t$ of $t$ `chan` essentially occurs covariantly (when used in `receive`) as well as contravariantly (when used in `send`); hence we must require that $C \vdash t \equiv t'$ in order for $C \vdash t$ `chan` $\subseteq t'$ `chan` to hold.

The definition of $C \vdash b_1 \subseteq b_2$ is a fairly straightforward axiomatisation of set inclusion upon behaviours; note that the premise for $C \vdash \{t_1 \text{ CHAN}\} \subseteq \{t_2 \text{ CHAN}\}$ is that $C \vdash t_1 \equiv t_2$.

### 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[4] 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 [18] 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 1.** 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 $C \vdash S_0\,C_0$.

A type $t$ is trivially solvable from $C$, and an environment $A$ is solvable from $C$ if for all $x$ in $Dom(A)$ it holds that $A(x)$ is solvable from $C$.

Except for the well-formedness requirement (explained later), rule (gen) seems close to the corresponding rule in [18]: clearly we cannot generalise over variables free in the global type assumptions or global constraint set, and as in effect systems (e.g. [19]) we cannot generalise over variables visible in the effect. Furthermore, as in [18] 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.* 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

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

[4] We use $\gamma$ to range over $\alpha$'s and $\beta$'s as appropriate and use $g$ range over $t$'s and $b$'s as appropriate.

and note that it is semantically unsound (at least if "$\cdots$" forked some process receiving twice over `ch` and adding the results). Writing $C = \{\{\alpha \text{ CHAN}\} \subseteq \beta, \{\{\texttt{int CHAN}\}\} \subseteq \beta, \{\{\texttt{bool CHAN}\}\} \subseteq \beta\}$ 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 $\forall(\alpha' : C').\ \alpha'$ `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 10 and Lemma 11) 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; actually the example suggests that if there is a constraint $(\{\alpha' \text{ CHAN}\} \subseteq \beta)$ then one should not generalise over $\alpha'$ if it is impossible to generalise over $\beta$. □

### 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. Motivated by the concluding remark of Example 2 we establish a relation between the left hand side variables and the right hand side variables of a constraint:

**Definition 2.** 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 $i = 1, 2$.

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:

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

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 3.** *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 4 below) and in ensuring that subtyping interacts well with the arrow relation (see Lemma 5 below).

**Lemma 4.** 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 $t' = t'_1$ `com` $b'$ there exist $t_1$ and $b$ such that $t = t_1$ `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 exists $t_1$ such that $t = t_1$ `chan` and such that $C \vdash t_1 \equiv t'_1$.
- If $t' = t'_1$ `list` there exists $t_1$ 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.

**Lemma 5.** Suppose $C$ is well-formed:

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

*Proof.* See Appendix A.

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 the situation in Example 2). The key idea is that if $C \vdash \gamma_1 \leftarrow \gamma_2$ then we do not generalise over $\gamma_1$ unless we also generalise over $\gamma_2$. These considerations lead to:

**Definition 6.** *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 $\{\vec{\alpha}\vec{\beta}\} = \{\vec{\alpha}\vec{\beta}\}^{C_0\uparrow}$.

A type $t$ is trivially well-formed.

Notice that if $C = \emptyset$ then $\forall(\vec{\alpha}\vec{\beta} : C).\ t$ is well-formed. It is essential for our development that the following property holds:

**Fact 7.** *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 $\{\vec{\alpha}\vec{\beta}\}$.

*Example 3.* Continuing Example 2 note that $\{\alpha'\}^{C'\uparrow} = \{\alpha', \beta\}$ showing that our current notion of well-formedness prevents the erroneous typing. □

*Example 4.* Continuing Example 1 we shall now briefly explain why it is accepted by our system. For this let us assume that `y` will have type $\alpha_y$ and that `x` will have type $\alpha_x$. Then the locally defined function

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

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

Now consider a system with subeffecting only (as in [23]): then for the type of `f` to match that of the locally defined function we have to give `f` the type $\alpha_x \to^b \alpha_x$ where $b = \{\alpha_y\ \text{CHAN}\}$. This then means that while the defining expression for `id` still has the type $\alpha_y \to^\emptyset \alpha_y$ we are unable to generalise it to $\forall(\alpha_y : \emptyset).\ \alpha_y \to^\emptyset \alpha_y$ because $\alpha_y$ 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 $\alpha_y$ and hence would allow to type the self-application of `id`.)

The system of [19] does not have subtyping but nevertheless the application of `id` to itself is typeable [19, section 11, the case (`id4 id4`)]. This is due to the

presence of *regions* and *masking* (cf. the discussion in the Introduction): with $\rho$ the region in which the new channel is allocated, the expression `sync (send (channel (), y))` does not contain $\rho$ in its type $\alpha_y$ and neither is $\rho$ present in the environment, so it is possible to discard the effect $\{\alpha_y \text{ CHAN}\}_\rho$ (recording that a channel carrying values of type $\alpha_y$ has been allocated in region $\rho$). Thus the two branches of the conditional can both be given type $\alpha_x \to^\emptyset \alpha_x$.

Also in the approach of [21] one can generalise over $\alpha_y$ and hence type the self-application of `id`. To see this, first note that $\alpha_y$ is classified as an imperative type variable (rather than an applicative type variable which would directly have allowed the generalisation) because $\alpha_y$ is used in the channel construct and thus has a side effect. Despite of this, next note that the 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 [21] 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 8.** For all constants $c$ of Figure 1, the type scheme TypeOf($c$) is closed, well-formed and solvable from $\emptyset$.

**Fact 9.** If $C, A \vdash e : \sigma \,\&\, b$ and $A$ is well-formed and solvable from $C$ then $\sigma$ is well-formed and solvable from $C$.

*Proof.* A straightforward case analysis on the last rule applied; for constants we make use of Fact 8.

**Lemma 10.** *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 11.** *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.

### 2.5 Proof Normalisation

It turns out that the proof of semantic soundness [2] 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[5]; to this end we define the notions of "normalised" and "strongly normalised" inference trees. But first we define an auxiliary concept:

**Definition 12.** An inference 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 13.** 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 11 in the cases (app), (let) and (if).

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

**Definition 14.** *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 $\sigma$ is a type) or TS-normalised (if $\sigma$ is a type scheme).

Notice that if $jdg = C, A \vdash e : \sigma \,\&\, b$ occurs in a normalised inference tree then we in fact have $jdg = C, A \vdash_n e : \sigma \,\&\, b$, unless $jdg$ is created by (con) or (id) and $\sigma$ is a type scheme. Next consider the result of applying (ins) to a normalised inference tree; even though the resulting inference tree is not normalised the resulting judgement in fact has a normalised inference:

**Lemma 15.** Suppose that

$$jdg = C, A \vdash e : S\, t_0 \,\&\, b$$

follows by an application of (ins) to the normalised judgement

[5] This will also facilitate proving the completeness of an inference algorithm.

$jdg' = C, A \vdash_n e : \forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0 \,\&\, b$

where $Dom(S) \subseteq \{\vec{\alpha}\vec{\beta}\}$ and $C \vdash S\,C_0$. Then also $jdg$ has a normalised inference:

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

*Proof.* The normalised judgement $jdg'$ follows by an application of (gen) to the normalised judgement

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

where $\{\vec{\alpha}\vec{\beta}\} \cap FV(C, A, b) = \emptyset$. From Lemma 10 we therefore get

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

and using Lemma 11 we get $C, A \vdash_n e : S\,t_0 \,\&\, b$ as desired.

It is not a severe restriction to work with normalised inferences only:

**Lemma 16.** *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 17.** An inference tree for $C, A \vdash e : \sigma \,\&\, b$ is strongly normalised if it is normalised as well as constraint-saturated. We write $C, A \vdash_s e : \sigma \,\&\, b$ when this is the case.

By Fact 13 we have:

**Fact 18.** A normalised inference tree $C, A \vdash_n e : \sigma \,\&\, b$ can be transformed into one that is strongly normalised.

### 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 $\alpha$, 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 $\forall\vec{\alpha}.u$.

From a type $t$ we can in a natural way construct an ML type $\epsilon(t)$: $\epsilon(\alpha) = \alpha$, $\epsilon(\texttt{int}) = \texttt{int}$, $\epsilon(t_1 \to^b t_2) = \epsilon(t_1) \to \epsilon(t_2)$, $\epsilon(t_1 \times t_2) = \epsilon(t_1) \times \epsilon(t_2)$, $\epsilon(t_1\ \texttt{list}) = \epsilon(t_1)\ \texttt{list}$, and $\epsilon(t\ \texttt{com}\ b) = \epsilon(t\ \texttt{chan}) = \epsilon(t)$.

From an ML type $u$ we can construct a type $\iota(u)$ by annotating all arrows with $\emptyset$ : $\iota(\alpha) = \alpha$, $\iota(\texttt{int}) = \texttt{int}$, $\iota(u_1 \rightarrow\ u_2) = \iota(u_1) \rightarrow^{\emptyset}\ \iota(u_2)$, $\iota(u_1 \times u_2) = \iota(u_1) \times \iota(u_2)$, $\iota(u_1\ \texttt{list}) = \iota(u_1)\ \texttt{list}$.

We say that a type scheme $ts = \forall(\vec{\alpha}\,\vec{\beta} : C).\ t$ is *sequential* if $C$ is empty. 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.) From an ML type scheme $us = \forall\vec{\alpha}\,.u$ we construct a (sequential) type scheme $\iota(us)$ as follows: $\iota(us) = \forall(\vec{\alpha} : \emptyset).\ \iota(u)$.

**Fact 19.** Let $u$ be an ML type, then $\epsilon(\iota(u)) = u$. Similarly for ML type schemes.

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`) $= \forall\alpha_1\alpha_2.\alpha_1 \rightarrow\ \alpha_2 \rightarrow\ \alpha_1 \times \alpha_2$.

**Assumption 20.** For all sequential constants $c$: $\iota(\text{MLTypeOf}(c)) = \text{TypeOf}(c)$ (so by Fact 19: $\text{MLTypeOf}(c) = \epsilon(\text{TypeOf}(c))$).

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

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

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

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

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

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

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

**Fig. 4.** The core of the ML type inference system.

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

**Theorem 21.** Let $e$ be a sequential expression:

- if $[\,] \vdash_{\text{ML}}\ e\ :\ u$ then $\emptyset, [\,] \vdash e\ :\ \iota(u)\,\&\,\emptyset$;
- if $C, [\,] \vdash e\ :\ t\,\&\,b$ where $C$ contains no type constraints then $[\,] \vdash_{\text{ML}}\ e\ :\ \epsilon(t)$.

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; in the future we hope to incorporate aspects of the causality information for the communication structure of Concurrent ML [15]. 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: Type and behaviour reconstruction for higher-order concurrent programs. To appear in *Journal of Functional Programming*, 1997.
2. T. Amtoft, F. Nielson, H.R. Nielson, J. Ammann: Polymorphic subtypes for effect analysis: the dynamic semantics. This volume of SLNCS, 1997.
3. L. Damas and R. Milner: Principal type-schemes for functional programs. In *Proc. of POPL '82*. ACM Press, 1982.
4. Y.-C. Fuh and P. Mishra: Polymorphic subtype inference: closing the theory-practice gap. In *Proc. TAPSOFT '89*. SLNCS 352, 1989.
5. Y.-C. Fuh and P. Mishra: Type inference with subtypes. *Theoretical Computer Science*, 73, 1990.
6. F. Henglein: Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253-289, 1993.
7. F. Henglein and C. Mossin: Polymorphic binding-time analysis. In *Proc. ESOP '94*, pages 287–301. SLNCS 788, 1994.
8. M.P. Jones: A theory of qualified types. In *Proc. ESOP '92*, pages 287–306. SLNCS 582, 1992.
9. X. Leroy and P. Weis: Polymorphic type inference and assignment. In *Proc. POPL '91*, pages 291–302. ACM Press, 1991.
10. R. Milner: A theory of type polymorphism in programming. *Journal of Computer Systems*, 17:348–375, 1978.
11. J.C. Mitchell: Type inference with simple subtypes. *Journal of Functional Programming*, 1(3), 1991.

12. F. Nielson and H.R. Nielson: Constraints for polymorphic behaviours for Concurrent ML. In *Proc. CCL'94*. SLNCS 845, 1994.
13. F. Nielson, H.R. Nielson, T. Amtoft: Polymorphic subtypes for effect analysis: the algorithm. This volume of SLNCS, 1997.
14. H.R. Nielson and F. Nielson: Automatic binding analysis for a typed $\lambda$-calculus. *Science of Computer Programming*, 10:139–176, 1988.
15. H.R. Nielson and F. Nielson: Higher-order concurrent programs with finite communication topology. In *Proc. POPL'94*, pages 84–97. ACM Press, 1994.
16. P. Panangaden and J.H. Reppy: The essence of Concurrent ML. In *ML with Concurrency: Design, Analysis, Implementation and Application* (editor: Flemming Nielson), Springer-Verlag, 1996.
17. J.H. Reppy: Concurrent ML: Design, application and semantics. In *Proc. Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198. SLNCS 693, 1993.
18. 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.
19. J.P. Talpin and P. Jouvelot: The type and effect discipline. *Information and Computation*, 111, 1994.
20. Y.-M. Tang: Control flow analysis by effect systems and abstract interpretation. PhD thesis, Ecoles des Mines de Paris, 1994.
21. M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.
22. M. Tofte and L. Birkedal: Region-annotated types and type schemes, 1996. Submitted for publication.
23. A.K. Wright: Typing references by effect inference. In *Proc. ESOP '92*, pages 473–491. SLNCS 582, 1992.

# A Proofs of Main Results

### Well-formedness

**Lemma 4** 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 $t' = t'_1$ `com` $b'$ there exist $t_1$ and $b$ such that $t = t_1$ `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 exists $t_1$ such that $t = t_1$ `chan` and such that $C \vdash t_1 \equiv t'_1$.
- If $t' = t'_1$ `list` there exists $t_1$ 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`).

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[6].

(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 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. □

For variables we need a different kind of lemma:

**Lemma 22.** 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 4 on the inference $C \vdash \alpha \subseteq t''$ we infer that $t''$ is a variable $\alpha''$. 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 $\alpha \in \{\alpha'\}^{C\downarrow}$. □

**Lemma 5** 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.

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

(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 $t'$:

$t' = t'_1 \to^{b'} t'_2$: 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 4 (applied to the former inference) tells us that there exist $t_1$, $b$ and $t_2$ such that $t = t_1 \to^b t_2$ 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 prior to the proof). Lemma 4 (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 $\to$: we can proceed as above.

$t'$ is a variable: Since $C \vdash t' \subseteq t$ we can use Lemma 4 to infer that $t$ is a variable; then we use Lemma 22 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 10** 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 Fig. 1 are closed (Fact 8). 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 $\mathit{Dom}(S_0) \subseteq \{\vec{\alpha}\vec{\beta}\}$, and wlog. we can assume that $\{\vec{\alpha}\vec{\beta}\}$ is disjoint from $\mathit{Inv}(S)$. 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 $\mathit{FV}(t_0, C_0)$ 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

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

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

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

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

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

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

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

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 7. 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 $\mathit{FV}(C_0)$ the result follows. Finally (7) holds trivially by choice of $\vec{\alpha'}\vec{\beta'}$. □

**Lemma 11** 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 cases (sub) and (ins) we use (a) together with the induction hypothesis.

**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

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

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

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

We now use a small trick: let $R$ be a renaming of the variables of $\{\vec{\alpha}\vec{\beta}\} \cap \mathit{FV}(C')$ to fresh variables. From $C' \vdash C$ and Lemma 10(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

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

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

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 $\mathit{Dom}(R') = \mathit{Ran}(R)$ and $R'\,\gamma' = \gamma$ if $R\,\gamma = \gamma'$ and $\gamma' \in \mathit{Dom}(R')$. Using Lemma 10(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 $\{\vec{\alpha}\vec{\beta}\} \cap \mathit{FV}(R\,C') = \emptyset$. □

**Proof normalisation**

**Lemma 16** 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 13, 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 $\mathit{Dom}(R) \subseteq \{\vec{\alpha}\vec{\beta}\}$ 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 8), and furthermore $\{\vec{\alpha'}\vec{\beta'}\} \cap \mathit{FV}(C, A, \emptyset) = \emptyset$ 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:

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

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 $C, A \vdash_c \texttt{fn}\ x \Rightarrow e \,:\, t_1 \to^b\ t_2 \,\&\, \emptyset$ 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 $C, A \vdash_n \texttt{fn}\ x \Rightarrow e \,:\, t_1 \to^b\ t_2 \,\&\, \emptyset$.

**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 $C, A \vdash_c \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \,:\, t_2 \,\&\, (b_1 \cup b_2)$ 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 9 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 we have $C, A \vdash_c e \,:\, S\,t_0 \,\&\, b$ 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 $C, A \vdash_n e \,:\, \forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0 \,\&\, b$ so by Lemma 15 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

**Theorem 21** Let $e$ be a sequential expression:

- if $[\,] \vdash_{\mathrm{ML}} e \,:\, u$ then $\emptyset, [\,] \vdash e \,:\, \iota(u) \,\&\, \emptyset$;
- if $C, [\,] \vdash e \,:\, t \,\&\, b$ where $C$ contains no type constraints then $[\,] \vdash_{\mathrm{ML}} e \,:\, \epsilon(t)$.

Before embarking on the proof we need to extend $\epsilon()$ and $\iota()$ to work on substitutions: from a substitution $S$ we construct an ML substitution $R = \epsilon(S)$ by stipulating $R\,\alpha = \epsilon(S\,\alpha)$; and from an ML substitution $R$ we construct a substitution $S = \iota(R)$ by stipulating $S\,\alpha = \iota(R\,\alpha)$.

**Fact 23.** For all $S$ and $R$ and all $t$ and $u$, we have $\epsilon(S\,t) = \epsilon(S)\,\epsilon(t)$ and $\iota(R\,u) = \iota(R)\,\iota(u)$.

*Proof.* The claims are proved by induction in $t$ respectively in $u$. If $t = \alpha$ the first equation follows from the definition of $\epsilon(S)$, and if $u = \alpha$ the second equation follows from the definition of $\iota(R)$. If $t$ (respectively $u$) is a base type like `int`, the equations are trivial. If $t$ is a composite type like $t_1 \to^b t_2$, the first 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 $u$ is a composite type like $u_1 \to u_2$, the second equation reads

$$\iota(R\,u_1) \to^\emptyset \iota(R\,u_2) = \iota(R)\,\iota(u_1) \to^\emptyset \iota(R)\,\iota(u_2)$$

and follows from the induction hypothesis. □

**Proof of the first part of Theorem 21** The first part of the theorem follows from the following proposition which admits a proof by induction:

**Proposition 24.** Let $e$ be sequential. If $A \vdash_{\mathrm{ML}} e : u$ then $\emptyset, \iota(A) \vdash e : \iota(u)\,\&\,\emptyset$. Similarly with $us$ instead of $u$.

*Proof.* Induction in the proof tree for $A \vdash_{\mathrm{ML}} e : us$, where we perform case analysis on the definition in Fig. 4 (the clauses for conditionals and for recursion are omitted, as they present no further complications).

**The case (con):** Follows from Assumption 20.

**The case (id):** Follows trivially.

**The case (abs):** By induction we get

$$\emptyset, \iota(A)[x : \iota(u_1)] \vdash e : \iota(u_2)\,\&\,\emptyset$$

and by using (abs) we get

$$\emptyset, \iota(A) \vdash \mathtt{fn}\ x \Rightarrow e : \iota(u_1) \to^\emptyset \iota(u_2)\,\&\,\emptyset$$

which is as desired since $\iota(u_1 \to u_2) = \iota(u_1) \to^\emptyset \iota(u_2)$.

**The case (app):** We can apply the induction hypothesis twice to arrive at

$$\emptyset, \iota(A) \vdash e_1 \,:\, \iota(u_2 \to\ u_1) \,\&\, \emptyset$$
$$\emptyset, \iota(A) \vdash e_2 \,:\, \iota(u_2) \,\&\, \emptyset$$

so since $\iota(u_2 \to\ u_1) = \iota(u_2) \to^{\emptyset}\ \iota(u_1)$ we can apply (app) to get

$$\emptyset, \iota(A) \vdash e_1\, e_2 \,:\, \iota(u_1) \,\&\, \emptyset \cup \emptyset \cup \emptyset.$$

As $\emptyset \vdash \emptyset \cup \emptyset \cup \emptyset \subseteq \emptyset$ by (sub) we have the desired judgement

$$\emptyset, \iota(A) \vdash e_1\, e_2 \,:\, \iota(u_1) \,\&\, \emptyset.$$

**The case (let):** By applying the induction hypothesis twice we get

$$\emptyset, \iota(A) \vdash e_1 \,:\, \iota(us_1) \,\&\, \emptyset$$
$$\emptyset, \iota(A)[x : \iota(us_1)] \vdash e_2 \,:\, \iota(u_2) \,\&\, \emptyset$$

so by applying (let) and (sub) we get the desired judgement

$$\emptyset, \iota(A) \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \,:\, \iota(u_2) \,\&\, \emptyset.$$

**The case (ins):** We can apply the induction hypothesis to get

$$\emptyset, \iota(A) \vdash e \,:\, \forall(\vec{\alpha} \,:\, \emptyset).\ \iota(u) \,\&\, \emptyset$$

so applying (ins) with the substitution $\iota(R)$ we arrive at

$$\emptyset, \iota(A) \vdash e \,:\, \iota(R)\, \iota(u) \,\&\, \emptyset$$

which is as desired since by Fact 23 we have $\iota(R)\, \iota(u) = \iota(R\, u)$.

**The case (gen):** We can apply the induction hypothesis to get

$$\emptyset, \iota(A) \vdash e \,:\, \iota(u) \,\&\, \emptyset$$

and the conclusion we want to arrive at is

$$\emptyset, \iota(A) \vdash e \,:\, \forall(\vec{\alpha} \,:\, \emptyset).\ \iota(u) \,\&\, \emptyset$$

which follows by using (gen) provided that (i) $\forall(\vec{\alpha} \,:\, \emptyset).\ \iota(u)$ is well-formed and solvable from $\emptyset$ and (ii) $\{\vec{\alpha}\,\} \cap FV(\iota(A)) = \emptyset$. Here (i) is trivial; and (ii) follows from $\{\vec{\alpha}\,\} \cap FV(A) = \emptyset$ since clearly $FV(\iota(A)) = FV(A)$. □

**Auxiliary notions.** Before embarking on the second part of Theorem 21 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 25.** 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 26.** 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 25 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 $\phi$ 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 $\phi$ 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 $\alpha$) 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 27.** 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, 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

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

**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 28.** 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 $FV(ts)$. (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 29.** 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 30.** We write $us \cong_\epsilon^R ts$ to mean that (for all $u$) $u \prec us$ iff $u \prec_\epsilon^R ts$.

**Fact 31.** Suppose $us = \epsilon(ts)$, where $ts = \forall(\vec{\alpha}\,\vec{\beta} : \emptyset).\ t$ is sequential. Then $us \cong_\epsilon^{\mathrm{Id}} ts$.

*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 $u = R\,\epsilon(t) \Leftrightarrow u \prec_\epsilon^{\mathrm{Id}} ts$. □

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

**Definition 32.** We write $u \cong_\epsilon^R t$ iff $u = R\,\epsilon(t)$.

**Definition 33.** We write $A' \cong_\epsilon^R A$ iff $Dom(A') = Dom(A)$ and $A'(x) \cong_\epsilon^R A(x)$ for all $x \in Dom(A)$.

**Fact 34.** Let $R$ and $S$ be such that $\epsilon(S) = R$ (this will for instance be the case if $S = \iota(R)$, cf. Fact 19). Then the relation $u \prec_\epsilon^R ts$ holds iff the relation $u \prec_\epsilon^{\mathrm{Id}} S\,ts$ holds.

Consequently, $us \cong_\epsilon^R ts$ holds iff $us \cong_\epsilon^{\mathrm{Id}} S\,ts$ holds.

*Proof.* Let $ts = \forall(\vec{\alpha}\,\vec{\beta} : C).\ t$. Due to the remark after Definition 28 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 23 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 $u \prec_\epsilon^R ts$.

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 $\mathit{Dom}(R) \cap \{\vec{\alpha}\,\} = \emptyset$, and if $\alpha \notin \vec{\alpha}$ we have $R''\,R\,\alpha = R\,\alpha = R'\,\alpha$ where the first equality sign follows from $\mathit{Ran}(R) \cap \{\vec{\alpha}\,\} = \emptyset$ and $\mathit{Dom}(R'') \subseteq \vec{\alpha}$. □

**Fact 35.** If $us \cong^{\mathrm{Id}}_{\epsilon} ts$ then $\mathit{FV}(us) \subseteq \mathit{FV}(ts)$.

*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 $\alpha_1$ be given such that $\alpha_1 \notin \mathit{FV}(ts)$, our task is to show that $\alpha_1 \notin \mathit{FV}(us)$.

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

**Proof of the second part of Theorem 21** The second part of the theorem follows (by letting $R = \mathrm{Id}$ and $A = A' = [\,]$) from the following proposition, which admits a proof by induction.

**Proposition 36.** 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 of $C, A \vdash e : ts\,\&\,b$, using terminology from Fig. 2 (the clauses for conditionals and for recursion are omitted, as they present no further complications):

**The case (con):** Here $ts = \mathrm{TypeOf}(c)$ and we choose $us = \mathrm{MLTypeOf}(c)$. By Assumption 20 and by Fact 31 we know that $us \cong^{\mathrm{Id}}_{\epsilon} ts$. Since $\mathrm{TypeOf}(c)$ is closed (cf. Fact 8) we have $us \cong^{\mathrm{Id}}_{\epsilon} \iota(R)\,ts$ so Fact 34 gives us the desired relation $us \cong^{R}_{\epsilon} ts$.

**The case (id):** Here $ts = A(x)$. Since $A' \cong^{R}_{\epsilon} A$ it holds that $A'(x) \cong^{R}_{\epsilon} A(x)$ 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

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

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 $A' \vdash_{\mathrm{ML}} \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : R\,\epsilon(t_2)$.

**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 27 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 27 tells us that $R$ satisfies $\epsilon(S_0\, C_0)$ which by Fact 23 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 23.

**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 $\{\vec{\alpha}\,\} \cap (\mathit{Dom}(R) \cup \mathit{Ran}(R)) = \emptyset$.

> Let $T$ be a renaming substitution mapping $\vec{\alpha}$ into fresh variables $\vec{\alpha}\,'$. By applying Lemma 10, 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 10) is equal to the shape of the original proof tree.

There exists $S_0$ with $\mathit{Dom}(S_0) \subseteq \{\vec{\alpha}\,\vec{\beta}\,\}$ such that $C \vdash S_0\, C_0$. Fact 27 then tells us that $R$ satisfies $\epsilon(S_0\, C_0)$ which by Fact 23 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 26 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 $S = \iota(R)$ (which by Fact 19 implies that $R = \epsilon(S)$); by Fact 34 and Fact 35 we infer that $FV(A') \subseteq FV(S\,A)$, so since $\{\vec{\alpha}\} \cap FV(A) = \emptyset$ and $\{\vec{\alpha}\} \cap Ran(S) = \emptyset$ we have $\{\vec{\alpha}\} \cap FV(A') = \emptyset$. 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 34 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 23; 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 $Dom(R_2) \subseteq \{\alpha\}$.