# Polymorphic Subtyping for Effect Analysis: the Algorithm

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

Computer Science Department, Aarhus University, Denmark

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

April 19, 1996

**Abstract**

We study an annotated type and effect system that integrates `let`-polymorphism, effects, and subtyping into an annotated type and effect system for a fragment of Concurrent ML. First a type inference algorithm and a procedure for constraint normalisation and simplification are defined, and next they are proved syntactically sound with respect to the annotated type and effect system.

## 1 Introduction

In a recent paper [8] we developed an annotated type and effect system for a fragment of Concurrent ML and in the companion paper [1] we proved it semantically sound. We now consider the algorithmic implications of the annotated type and effect system that integrates ML-style polymorphism (the `let`-construct), subtyping (with the usual contravariant ordering for function spaces), and effects (for the set of "dangerous variables").

The previous papers already mentioned one key idea as far as the annotated type and effect system is concerned, and this is now supplemented by an analogous key idea concerning the construction of the algorithm; the two key ideas are:

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

- Defining the set of variables over which to generalise in the algorithm; this involves taking downwards as well as simultaneous upwards and downwards closures with respect to a constraint set and is essential for achieving syntactic soundness (and eventually syntactic completeness).

In this paper we develop an algorihm $\mathcal{W}$ for producing the typings of a given program; it is constructed by means of a syntax directed algorithm $\mathcal{W}'$, an algorithm $\mathcal{F}$ for ensuring that constraints are well-formed, and an (optional) algorithm $\mathcal{R}$ for a rather dramatic reduction in the size of constraint sets. We prove that the algorithms are syntactically sound and the issue of completeness seems promising.

We shall see that the algorithm generates a set of type and behaviour constraints that can always be solved provided recursive behaviour systems are admitted. Alternatively recursive behaviour systems can be disallowed thus rejecting programs that implement recursion in an indirect way through communication; this is quite analogous to the way the absence of recursive types in the simply typed $\lambda$-calculi forbids defining the $Y$ combinator and instead requires recursion to be an explicit primitive in the language.

## 2 Inference System

In this section we briefly recapitulate the inference system presented in [8]. Expressions and constants are given by

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

where there are four kinds of constants: sequential constructors like `true` and `pair`, sequential base functions like `+` and `fst`, the non-sequential constructors `send` and `receive`, and the non-sequential base functions `sync`, `channel` and `fork`.

Types and behaviours are given by

$$\begin{array}{lcl} t & ::= & \alpha \mid \texttt{unit} \mid \texttt{int} \mid \texttt{bool} \mid t_1 \times t_2 \mid t\ \texttt{list} \\ & \mid & t_1 \rightarrow^b t_2 \mid t\ \texttt{chan} \mid t\ \texttt{com}\ b \\ b & ::= & \{t\ \text{CHAN}\} \mid \beta \mid \emptyset \mid b_1 \cup b_2 \end{array}$$

Type schemes $ts$ are of form $\forall(\vec{\alpha}\vec{\beta} : C).\ t$ with $C$ a set of constraints, where a constraint is either of form $t_1 \subseteq t_2$ or of form $b_1 \subseteq b_2$. The type schemes of selected constants are given in Figure 1.

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

Figure 1: Type schemes for selected constants.

The ordering among types and behaviours is depicted in Figure 2; in particular notice that the ordering is contravariant in the argument position of a function type and that in order for $t$ `chan` $\subseteq t'$ `chan` and $\{t\ \text{CHAN}\} \subseteq \{t'\ \text{CHAN}\}$ to hold we must demand that $t \equiv t'$, i.e. $t \subseteq t'$ and $t' \subseteq t$, since $t$ occurs covariantly when used in `receive` and contravariantly when used in `send`.

The inference system is depicted in Figure 3 and employs the notion of well-formedness:

**Definition 2.1** A constraint set is well-formed if all constraints are of form $t \subseteq \alpha$ or $b \subseteq \beta$; and a type scheme $\forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0$ is well-formed if $C_0$ is well-formed and if all constraints in $C_0$ contain at least one variable among $\{\vec{\alpha}\vec{\beta}\}$ and if $\{\vec{\alpha}\vec{\beta}\}^{C_0\uparrow} = \{\vec{\alpha}\vec{\beta}\}$. □

Here[1] we make use of *upwards closure* defined as follows:

$$X^{C\uparrow} = \{\gamma \mid \exists\gamma' \in X : C \vdash \gamma' \leftarrow^{*} \gamma\}$$

where 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 $i = 1, 2$, and where we use $\leftarrow^{*}$ for the reflexive and transitive closure. In a similar way we define

$$X^{C\downarrow} = \{\gamma \mid \exists\gamma' \in X : C \vdash \gamma \leftarrow^{*} \gamma'\} \text{ and}$$
$$X^{C\updownarrow} = \{\gamma \mid \exists\gamma' \in X : C \vdash \gamma' \leftrightarrow^{*} \gamma\}$$

[1]We use $g$ to range over $t$ or $b$ as appropriate and $\gamma$ to range over $\alpha$ and $\beta$ as appropriate and $\sigma$ to range over $t$ and $ts$ as appropriate.

## Ordering on behaviours

(axiom) $C \vdash b_1 \subseteq b_2$ if $(b_1 \subseteq b_2) \in C$

(refl) $C \vdash b \subseteq b$

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

(CHAN) $$\frac{C \vdash t \equiv t'}{C \vdash \{t\ \textsc{chan}\} \subseteq \{t'\ \textsc{chan}\}}$$

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

( $\cup$ ) $C \vdash b_i \subseteq (b_1 \cup b_2)$ for $i = 1, 2$

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

## Ordering on types

(axiom) $C \vdash t_1 \subseteq t_2$ if $(t_1 \subseteq t_2) \in C$

(refl) $C \vdash t \subseteq t$

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

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

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

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

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

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

Figure 2: Subtyping and subeffecting.

(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 3: The type inference system.

where the relation $\leftrightarrow$ is the *union* of $\leftarrow$ and $\rightarrow$, with $\rightarrow$ the inverse of $\leftarrow$. Also we write $C \vdash C_0$ to mean that $C \vdash g_1 \subseteq g_2$ for all $g_1 \subseteq g_2$ in $C_0$ and we say that the type scheme $\forall(\vec{\alpha}\vec{\beta} : C_0).\ t_0$ is solvable from $C$ by $S_0$ if $Dom(S_0) \subseteq \{\vec{\alpha}\vec{\beta}\}$ and if $C \vdash S_0\, C_0$.

## 2.1 Properties of the Inference System

In [8] we proved the lemmas below which express how to get valid judgements from valid judgements: we shall see that these results are crucial for showing soundness of our inference algorithm.

**Lemma 2.2** *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).

**Lemma 2.3** *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).

## 3 The Inference Algorithm

In designing an inference algorithm $\mathcal{W}$ for the type inference system we are motivated by the overall approach of [9, 3]. One ingredient (called $\mathcal{W}'$) of this will be to perform a syntax-directed traversal of the expression in order to determine its type and behaviour; this will involve constructing a constraint set for expressing the required relationship between the type and behaviour variables. The second ingredient (called $\mathcal{F}$) will be to perform a decomposition of the constraint set into one that is well-formed and that hopefully contains much fewer constraints. The third ingredient (called $\mathcal{R}$) amounts to further reducing the constraint set; this is optional and a somewhat open ended endeavour.

### 3.1 Well-formedness, Atomicity, and Simplicity

We need to introduce these three properties of constraint sets, types, type schemes, behaviours, assumption lists, and substitutions. Already in Definition 2.1 we introduced the notion of well-formedness for constraint sets and type schemes; in [8] it was argued that this notion is essential for the semantic soundness of the inference system and this claim was substantiated in [1]. In addition we stipulate:

**Definition 3.1** Types, behaviours, and substitutions are trivially well-formed. An assumption list is well-formed if all its type schemes are.

**Definition 3.2** A constraint set is *atomic* if all $(t_1 \subseteq t_2)$ in the set have $t_1$ to be a type variable and if all $(b_1 \subseteq b_2)$ have $b_1$ to be a behaviour variable or a singleton $\{t \text{ CHAN}\}$; a type scheme is atomic if its constraint set is, and an assumption list is atomic if all its type schemes are; finally types, behaviours and substitutions are trivially atomic.

Atomicity of behaviour constraints is unproblematic because a constraint $(\emptyset \subseteq b)$ can always be thrown away and a constraint $(b_1 \cup b_2 \subseteq b)$ can always be split to $(b_1 \subseteq b)$ and $(b_2 \subseteq b)$. Atomicity of well-formed type constraints is responsible for disallowing constraint like $(\texttt{int} \subseteq \alpha)$ and $(t_1 \times t_2 \subseteq \alpha)$ by forcing $\alpha$ to be replaced by a type expression that "matches" the left hand side. This phenomenon can be found in [5, 3, 9] as well. It is responsible for making the algorithm a "conservative extension" (cf. [8]) of the way algorithm $\mathcal{W}$ for Standard ML would operate if effects were not taken into account: in particular our algorithm will fail, rather than produce an unsolvable constraint set, if the underlying type constraints of the effect-free system cannot be solved.

**Definition 3.3** A type is *simple* if all its behaviour annotations are behaviour variables; a behaviour is simple if all types occurring in it are simple; a constraint set is simple if all the types and behaviours occurring in it are simple *and* if all behaviour constraints $(b_1 \subseteq b_2)$ have the right hand side $(b_2)$ to be a variable; a type scheme is simple if the constraint set and the type both are; an assumption list is simple if all its type schemes are; finally a substitution is simple if it maps behaviour variables to behaviour variables (*rather than* simple behaviours) and type variables to simple types.

In examples we shall allow to weaken this restriction by allowing types to contain $\emptyset$ annotations in covariant positions; we shall then say that the type is *essentially simple* as it can easily be replaced (without changing the set of "instances") by a simple type that uses fresh behaviour variables instead of $\emptyset$.

**Fact 3.4** For all constants $c$, the type scheme $\text{TypeOf}(c)$ is essentially simple.

The notion of simplicity is taken from [11] and is used also in [7] and is a way to overcome the need for otherwise having to perform unification (or decomposition) in a non-free algebra (like the algebra of behaviours). It is a key technical assumption necessary for being able to maintain well-formedness of constraint sets as we have no techniques available for decomposing a constraint of form $\beta_1 \subseteq \beta_2 \cup \beta_3$ into a set of well-formed constraints and therefore we need to ensure that constraints of this form never arise in the algorithm.

**Fact 3.5** Let $t$ be a simple type, $b$ be a simple behaviour, $C$ be a simple constraint set, $ts$ be a simple type scheme, and $S$, $S'$ be simple substitutions. Then $S\,t$ is a simple type, $S\,b$ is a simple behaviour, $S\,C$ is a simple constraint set, $S\,ts$ is a simple type scheme, and $S'\,S$ is a simple substitution.

## 3.2 Algorithm $\mathcal{W}$

Our key algorithm $\mathcal{W}$ is described by

$$\mathcal{W}(A, e) = (S, t, b, C)$$

where the intuition is that $C, S\,A \vdash e : t \,\&\, b$ is the "best correct" typing of $e$ relative to an assumption list derived from $A$. We shall enforce throughout (by using $\mathcal{F}$) that all of $S$, $t$, $b$ and $C$ are well-formed, atomic and simple provided that $A$ is simple. Algorithm $\mathcal{W}$ is defined by the clause

$$\begin{aligned}\mathcal{W}(A, e) = {} & \text{let } (S_1, t_1, b_1, C_1) = \mathcal{W}'(A, e)\\ & \text{let } (S_2, C_2) = \mathcal{F}(C_1)\\ & \text{let } (C_3, t_3, b_3) = \mathcal{R}(C_2, S_2\, t_1, S_2\, b_1, S_2\, S_1\, A)\\ & \text{in } (S_2\, S_1, t_3, b_3, C_3)\end{aligned}$$

Here algorithm $\mathcal{W}'$ is defined in terms of algorithm $\mathcal{W}$ and is responsible for the syntax-directed traversal of the argument expression $e$. In general, $\mathcal{W}'$ will fail to produce a well-formed and atomic constraint set $C$, even when the assumption list $A$ is well-formed and simple; it will be the case, however, that all of $S_1$, $t_1$, $b_1$ and $C_1$ are simple (and that $S_1$, $t_1$, and $b_1$ are trivially well-formed and atomic). This then motivates the need for a transformation $\mathcal{F}$ (Section 4) that maps a simple constraint set into a simple, well-formed and atomic constraint set; since this involves splitting variables we shall need to produce a simple (and trivially well-formed and atomic) substitution as well. The final transformation $\mathcal{R}$ merely attempts to get a smaller constraint set by removing variables that are not strictly needed. Its operation is not essential for the soundness of our algorithm and thus one might define it by $\mathcal{R}(C, t, b, A) = (C, t, b)$; in Section 5 we shall consider a more powerful version of $\mathcal{R}$.

**Example 3.6** To make the intentions a bit clearer suppose that $\mathcal{W}'(A, e) = (S_1, t_1, b_1, C_1)$ so that $C_1, S_1\,A \vdash e : t_1 \,\&\, b_1$ is the "best correct" typing of $e$. If

$$C_1 = \{\alpha_1 \times \alpha_2 \subseteq \alpha_3,\ \texttt{int} \subseteq \alpha_4,\ \{\alpha_5\ \text{CHAN}\} \cup \emptyset \subseteq \beta\}$$

then $(S_2, C_2) = \mathcal{F}(C_1)$ should give

$$\begin{aligned} C_2 &= \{\alpha_1 \subseteq \alpha_{31},\ \alpha_2 \subseteq \alpha_{32},\ \{\alpha_5\ \text{CHAN}\} \subseteq \beta\}\\ S_2 &= [\alpha_3 \mapsto \alpha_{31} \times \alpha_{32},\ \alpha_4 \mapsto \texttt{int}]\end{aligned}$$

Here we expand $\alpha_3$ to $\alpha_{31} \times \alpha_{32}$ so that the resulting constraint $\alpha_1 \times \alpha_2 \subseteq \alpha_{31} \times \alpha_{32}$ can be "decomposed" into $\alpha_1 \subseteq \alpha_{31}$ and $\alpha_2 \subseteq \alpha_{32}$ that are both well-formed and atomic. Furthermore we have expanded $\alpha_4$ to `int` as it follows from Figure 2 that $\emptyset \vdash \texttt{int} \subseteq t$ necessitates that $t$ equals `int`. Finally we have decomposed the constraint upon $\beta$ into two and then removed the trivial $\emptyset \subseteq \beta$ constraint. Clearly the intention is that also $C_2, S_2\, S_1\, A \vdash e \,:\, S_2\, t_1 \,\&\, S_2\, b_1$ is the "best correct" typing of $e$ and additionally the constraint set is well-formed and atomic (unlike what is the case for $C_1$). $\square$

## 3.3 Algorithm $\mathcal{W}'$

Algorithm $\mathcal{W}'$ is defined by the clauses in Figure 4 and is to be defined simultaneously with $\mathcal{W}$ since it calls $\mathcal{W}$ in a number of places. Actually it could call itself recursively, rather than calling $\mathcal{W}$, in all but one place[2]: the call to $\mathcal{W}$ immediately prior to the use of *GEN* to generalise the type of the `let`-bound identifier to a type scheme. The algorithm follows the overall approach of [9, 4] except that as in [3] there are no explicit unification steps; these all take place as part of the $\mathcal{F}$ transformation. The only novel ingredient of our approach shows up in the clause for `let` as we shall explain shortly. Concentrating on "the overall picture" we thus have clauses for identifiers and constants; both make use of the auxiliary function *INST* defined by

$$\begin{array}{lcl} \mathit{INST}(\forall(\vec{\alpha}\vec{\beta} : C).\ t) & = & \text{let } \vec{\alpha}'\vec{\beta}' \text{ be fresh} \\ & & \text{let } R = [\vec{\alpha}\vec{\beta} \mapsto \vec{\alpha}'\vec{\beta}'] \\ & & \text{in } (\mathrm{Id}, R\,t, \emptyset, R\,C) \\ & & \\ \mathit{INST}(t) & = & (\mathrm{Id}, t, \emptyset, \emptyset) \end{array}$$

in order to produce a fresh instance of the relevant type or type scheme (as determined from TypeOf or from $A$); if the constant or identifier is unknown, failure is reported. The clause for function abstraction is rather straightforward; note the use of a fresh behaviour variable in order to ensure that only simple types are produced; we then add a constraint to record the "meaning" of the behaviour variable. Also the clause for application is rather straightforward; note that instead of a unification step we record the desired connection between the operator and operand types by means of a constraint. The clauses for recursion and conditional follow the same pattern as the clauses for abstraction and application.

The only novelty in the clause for `let` is the function *GEN* used for generalisation:

$$\begin{array}{lcl} \mathit{GEN}(A,b)(C,t) = & & \text{let } \{\vec{\alpha}\vec{\beta}\} = (FV(t)^{C\updownarrow}) \setminus (FV(A,b)^{C\downarrow}) \\ & & \text{let } C_0 = C \mid_{\{\vec{\alpha}\vec{\beta}\}} \\ & & \text{in } \forall(\vec{\alpha}\vec{\beta} : C_0).\ t \end{array}$$

[2]Interestingly, this is exactly the place where the algorithm of [9] makes use of constraint simplification in the "*close*" function; however, our prototype implementation suggests that the choice embodied in the definition of $\mathcal{W}$ gives faster performance.

```
$\mathcal{W}'(A, c)$ = if $c \in Dom(\mathrm{TypeOf})$ then $INST(\mathrm{TypeOf}(c))$ else $fail_{const}$

$\mathcal{W}'(A, x)$ = if $x \in Dom(A)$ then $INST(A(x))$ else $fail_{ident}$

$\mathcal{W}'(A, \texttt{fn}\ x \Rightarrow e_0)$ =
        let $\alpha$ be fresh
        let $(S_0, t_0, b_0, C_0) = \mathcal{W}(A[x : \alpha], e_0)$
        let $\beta$ be fresh
        in $(S_0, S_0\,\alpha \to^{\beta} t_0, \emptyset, C_0 \cup \{b_0 \subseteq \beta\})$

$\mathcal{W}'(A, e_1\,e_2)$ =
        let $(S_1, t_1, b_1, C_1) = \mathcal{W}(A, e_1)$
        let $(S_2, t_2, b_2, C_2) = \mathcal{W}(S_1\,A, e_2)$
        let $\alpha, \beta$ be fresh
        in $(S_2\,S_1, \alpha, S_2\,b_1 \cup b_2 \cup \beta,$
            $S_2\,C_1 \cup C_2 \cup \{S_2\,t_1 \subseteq t_2 \to^{\beta} \alpha\})$

$\mathcal{W}'(A, \texttt{let}\ x = e_1\ \texttt{in}\ e_2)$ =
        let $(S_1, t_1, b_1, C_1) = \mathcal{W}(A, e_1)$
        let $ts_1 = GEN(S_1\,A, b_1)(C_1, t_1)$
        let $(S_2, t_2, b_2, C_2) = \mathcal{W}((S_1\,A)[x : ts_1], e_2)$
        in $(S_2\,S_1, t_2, S_2\,b_1 \cup b_2, S_2\,C_1 \cup C_2)$

$\mathcal{W}'(A, \texttt{rec}\ f\ x \Rightarrow e_0)$ =
        let $\alpha_1, \beta, \alpha_2$ be fresh
        let $(S_0, t_0, b_0, C_0) = \mathcal{W}(A[f : \alpha_1 \to^{\beta} \alpha_2][x : \alpha_1], e_0)$
        in $(S_0, S_0\,(\alpha_1 \to^{\beta} \alpha_2), \emptyset, C_0 \cup \{b_0 \subseteq S_0\,\beta,\ t_0 \subseteq S_0\,\alpha_2\})$

$\mathcal{W}'(A, \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2)$ =
        let $(S_0, t_0, b_0, C_0) = \mathcal{W}(A, e_0)$
        let $(S_1, t_1, b_1, C_1) = \mathcal{W}(S_0\,A, e_1)$
        let $(S_2, t_2, b_2, C_2) = \mathcal{W}(S_1\,S_0\,A, e_2)$
        let $\alpha$ be fresh
        in $(S_2\,S_1\,S_0, \alpha, S_2\,S_1\,b_0 \cup S_2\,b_1 \cup b_2,$
            $S_2\,S_1\,C_0 \cup S_2\,C_1 \cup C_2 \cup \{S_2\,S_1\,t_0 \subseteq \texttt{bool},\ S_2\,t_1 \subseteq \alpha,\ t_2 \subseteq \alpha\})$
```

Figure 4: Syntax-directed constraint generation.

where $C \mid_{\{\vec{\alpha}\vec{\beta}\}} = \{(g_1 \subseteq g_2) \in C \mid FV(g_1, g_2) \cap \{\vec{\alpha}\vec{\beta}\} \neq \emptyset\}$. The definition of $C_0$ thus establishes the part of the well-formedness condition that requires each constraint to involve at least one bound variable.

The exclusion of the set $FV(A,b)^{C\downarrow}$ (rather than just $FV(A,b)$) is necessary in order to ensure $\{\vec{\alpha}\vec{\beta}\}^{C\uparrow} = \{\vec{\alpha}\vec{\beta}\}$ which is essential for semantic soundness [8, 1]. Finally we have chosen $FV(t)^{C\updownarrow}$ as the "universe" in which to perform the set difference; this universe must be large enough that we will still get syntactic completeness and all of $FV(t)$, $FV(t)^{C\downarrow}$ (these two are not even upwards closed) and $FV(t)^{C\uparrow}$ would have been too small for this.

**Fact 3.7** Let $\sigma = GEN(A,b)(C,t)$.

(a) If $C$ is well-formed then so is $\sigma$.

(b) If $C$ and $t$ are simple, atomic and well-formed then so is $\sigma$.

**Proof** The only non-trivial task is to show that $\{\vec{\alpha}\vec{\beta}\}^{C\uparrow} \subseteq \{\vec{\alpha}\vec{\beta}\}$ where $\{\vec{\alpha}\vec{\beta}\}$ is as in the defining clause for $GEN$. So assume $C \vdash \gamma_1 \leftarrow \gamma_2$ with $\gamma_1 \in \{\vec{\alpha}\vec{\beta}\}$; we must show that $\gamma_2 \in \{\vec{\alpha}\vec{\beta}\}$. Now $\gamma_1 \in FV(t)^{C\updownarrow}$ and $\gamma_1 \notin FV(A,b)^{C\downarrow}$, hence we infer $\gamma_2 \in FV(t)^{C\updownarrow}$ and $\gamma_2 \notin FV(A,b)^{C\downarrow}$ which amounts to the desired result. $\square$

**Remark:** Note that $FV(t)^{C\updownarrow}$ is a subset of $FV(t,C)$ and that it may well be a proper subset; when this is the case it avoids to generalise over "purely internal" variables that are inconsequential for the overall type. If one were to regard `let` $x = e_1$ `in` $e_2$ as equivalent to $e_2[e_1/x]$ (which is only the case if $e_1$ has an empty behaviour) this corresponds to forcing all "purely internal" variables in corresponding copies of $e_1$ to be equal. This is helpful for reducing the size of constraint sets and type schemes. $\square$

## 4 Algorithm $\mathcal{F}$

The transformation $\mathcal{F}$ may be described as a non-deterministic rewriting process. It operates over triples of the form $(S, C, \sim)$ where $S$ is a substitution, $C$ is a constraint set, and $\sim$ is an equivalence relation among the finite set of type variables in $C$; we shall write $\mathrm{Eq}_C$ for the identity relation over type variables in $C$. We then define $\mathcal{F}$ by

$$\begin{array}{l} \mathcal{F}(C) = \text{let } (S', C', \sim') \text{ be given by } (\mathrm{Id}, C, \mathrm{Eq}_C) \longrightarrow^{*} (S', C', \sim') \not\longrightarrow \\ \qquad\quad \text{in if } C' \text{ is atomic and well-formed} \\ \qquad\qquad \text{then } (S', C') \text{ else } \mathit{fail}_{\mathit{forcing}} \end{array}$$

$(\emptyset)\quad C \dot\cup \{\emptyset \subseteq b\} \rightharpoonup C$

$(\cup)\quad C \dot\cup \{b_1 \cup b_2 \subseteq b\} \rightharpoonup C \cup \{b_1 \subseteq b, b_2 \subseteq b\}$

$(\times)\quad C \dot\cup \{t_1 \times t_2 \subseteq t_3 \times t_4\} \rightharpoonup C \cup \{t_1 \subseteq t_3, t_2 \subseteq t_4\}$

$(\texttt{list})\; C \dot\cup \{t_1\ \texttt{list} \subseteq t_2\ \texttt{list}\} \rightharpoonup C \cup \{t_1 \subseteq t_2\}$

$(\texttt{chan})\; C \dot\cup \{t_1\ \texttt{chan} \subseteq t_2\ \texttt{chan}\} \rightharpoonup C \cup \{t_1 \subseteq t_2, t_2 \subseteq t_1\}$

$(\texttt{com})\quad C \dot\cup \{t_1\ \texttt{com}\ b_1 \subseteq t_2\ \texttt{com}\ b_2\} \rightharpoonup C \cup \{t_1 \subseteq t_2, b_1 \subseteq b_2\}$

$$(\rightarrow)\quad C \dot\cup \{t_1 \rightarrow^{b_1} t_2 \subseteq t_3 \rightarrow^{b_2} t_4\}$$
$$\rightharpoonup C \cup \{t_3 \subseteq t_1, b_1 \subseteq b_2, t_2 \subseteq t_4\}$$

$$\left.\begin{array}{l} (\texttt{int})\ C \dot\cup \{\texttt{int} \subseteq \texttt{int}\} \\ (\texttt{bool})\ C \dot\cup \{\texttt{bool} \subseteq \texttt{bool}\} \\ (\texttt{unit})\ C \dot\cup \{\texttt{unit} \subseteq \texttt{unit}\} \end{array}\right\} \rightharpoonup C$$

Figure 5: Decomposition of constraints.

$$(\texttt{dc})\ \frac{C \rightharpoonup C'}{(S, C, \sim) \longrightarrow (S, C', \sim)}$$

$$(\texttt{mr})\ (S, C \dot\cup \{t \subseteq \alpha\}, \sim) \longrightarrow (R\,S, R\,C \cup \{R\,t \subseteq R\,\alpha\}, \sim')$$
$$\text{where } (R, \sim') = \mathcal{M}(\alpha, t, \sim)$$

$$(\texttt{ml})\ (S, C \dot\cup \{\alpha \subseteq t\}, \sim) \longrightarrow (R\,S, R\,C \cup \{R\,\alpha \subseteq R\,t\}, \sim')$$
$$\text{where } (R, \sim') = \mathcal{M}(\alpha, t, \sim)$$

Figure 6: Rewriting rules for $\mathcal{F}$: forcing well-formedness.

The rewriting relation is defined by the axioms of Figure 6 and will be explained below; it makes use of an auxiliary rewriting relation, defined in Figure 5, which operates over constraint sets.

The axioms of Figure 5 are rather straightforward. For behaviours the axiom $(\emptyset)$ simply throws away constraints of the form $\emptyset \subseteq b$ and the axiom $(\cup)$ simply decomposes constraints of the form $b_1 \cup b_2 \subseteq b$ to the simpler constraints $b_1 \subseteq b$ and $b_2 \subseteq b$. (A small notational point: in Figure 5 and in Figure 6 we write $C \dot{\cup} C'$ for $C \cup C'$ in case $C \cap C' = \emptyset$.) For types the axioms ( $\times$ ), (`list`), (`chan`), (`com`), and ($\rightarrow$) essentially run the inference system of Figure 2 in a backwards way and generate new constraints $t_1 \subseteq t_2$ and $t_2 \subseteq t_1$ whenever we had $t_1 \equiv t_2$ in Figure 2. Axioms (`int`), (`bool`) and (`unit`) are simple instances of reflexivity.

**Fact 4.1** The rewriting relation $\rightharpoonup$ is confluent and if $C_1 \rightharpoonup C_2$ then $C_2 \vdash C_1$.

**Proof** Confluence follows since each rewriting operates on a single element only, and for each element there is only one possible rewriting. □

We now turn to Figure 6. The axiom (dc) decomposes the constraint set but does not modify the substitution nor the equivalence relation among type variables. The axioms (mr) and (ml) force left and right hand sides of type constraints to match. This is related to unification and produces a new substitution as a result; additionally it may modify the equivalence relation among type variables. The details require the function $\mathcal{M}$ (which may be undefined when the "occur check" fails) to be defined shortly. Before presenting the formal definition we consider an example.

**Example 4.2** Consider the constraint $t_1 \subseteq \alpha_0$ where $t_1 = (\alpha_{11} \times \alpha_{12})\ \texttt{com}\ \beta_1$. Forcing the left and right hand sides to match means finding a substitution $R$ such that $R t_1$ and $R \alpha_0$ have the same shape. A natural way to achieve this is by creating new type variables $\alpha_{21}$ and $\alpha_{22}$ and a new behaviour variable $\beta_2$ and by defining

$$R = [\alpha_0 \mapsto (\alpha_{21} \times \alpha_{22})\ \texttt{com}\ \beta_2].$$

Then $R t_1 = t_1 = (\alpha_{11} \times \alpha_{12})\ \texttt{com}\ \beta_1$ and $R \alpha_0 = (\alpha_{21} \times \alpha_{22})\ \texttt{com}\ \beta_2$ and these types intuitively have the same shape. Returning to Figure 6 we would thus expect $\mathcal{M}(\alpha_0, t_1, \sim) = (R, \sim)$.

If instead we had considered the constraint $(\alpha \times \alpha)\ \texttt{com}\ \beta \subseteq \alpha$ then the above procedure would not lead to a matching constraint. We would get

$$R = [\alpha \mapsto (\alpha' \times \alpha'')\ \texttt{com}\ \beta']$$

and the constraint $R((\alpha \times \alpha)\ \texttt{com}\ \beta) \subseteq R\alpha$ then is

$$(((\alpha' \times \alpha'')\ \texttt{com}\ \beta') \times ((\alpha' \times \alpha'')\ \texttt{com}\ \beta'))\ \texttt{com}\ \beta \subseteq (\alpha' \times \alpha'')\ \texttt{com}\ \beta'$$

which does not match. Indeed it would seem that matching could go on forever without ever producing a matching result. To detect this situation we have an "occurs check": when $\mathcal{M}(\alpha, t, \sim) = (R, \sim')$ no variable in $Dom(R)$ must occur in $t$. This condition fails when $t = (\alpha \times \alpha)\ \mathtt{com}\ \beta$.

There are more subtle ways in which termination may fail. Consider the constraint set

$$\{\alpha_1\ \mathtt{com}\ \beta_1 \subseteq \alpha_0,\ \alpha_0 \subseteq \alpha_1\}$$

where only the first constraint does not match. Attempting a match we get

$$R_1 = [\alpha_0 \mapsto \alpha_2\ \mathtt{com}\ \beta_2]$$

and note that the "occurs check" succeeds. The resulting constraint set is

$$\{\alpha_1\ \mathtt{com}\ \beta_1 \subseteq \alpha_2\ \mathtt{com}\ \beta_2,\ \alpha_2\ \mathtt{com}\ \beta_2 \subseteq \alpha_1\}$$

which may be reduced to

$$\{\alpha_1 \subseteq \alpha_2,\ \beta_1 \subseteq \beta_2,\ \alpha_2\ \mathtt{com}\ \beta_2 \subseteq \alpha_1\}.$$

The type part is isomorphic to the initial constraints, so this process may continue forever: we perform a second match and produce a second substitution $R_2$, etc.

To detect this situation we as in [3] make use of the equivalence relation $\sim$ and extend it with $\alpha_1 \sim \alpha_2$ after the first match that produced $R_1$. When performing the second match we then require $R_2$ not only to expand $\alpha_1$ but also all $\alpha'$ satisfying $\alpha' \sim \alpha_1$; this means that $R_2$ must expand also $\alpha_2$. Consequently the "extended occurs check"

$$Dom(R_2) \cap FV(\alpha_2\ \mathtt{com}\ \beta_2) = \emptyset$$

fails. □

To formalise the development of the example we need to be more precise about the shape of a type and when two types match.

**Definition 4.3** A shape $sh$ is a type with holes in it for all type variables and for all behaviours; it may be formally defined by:

$$\begin{array}{rcl} sh & ::= & [\,]\ |\ \mathtt{unit}\ |\ \mathtt{int}\ |\ \mathtt{bool}\ |\ sh_1 \times sh_2\ |\ sh_1 \rightarrow^{[\,]} sh_2\ |\ sh\ \mathtt{list} \\ & | & sh\ \mathtt{chan}\ |\ sh\ \mathtt{com}\ [\,] \end{array}$$

```
$\mathcal{M}(\alpha, t, \sim) =$
        let $sh[\vec{\alpha}_0, \vec{b}_0] = t$
        let $\{\alpha_1, \cdots, \alpha_n\} = \{\alpha' \mid \alpha' \sim \alpha\}$
        let $\vec{\alpha}_1, \cdots, \vec{\alpha}_n$ be vectors of fresh variables
                        of the same length as $\vec{\alpha}_0$
        let $\vec{\beta}_1, \cdots, \vec{\beta}_n$ be vectors of fresh variables
                        of the same length as $\vec{b}_0$
        let $R = [\alpha_1 \mapsto sh[\vec{\alpha}_1, \vec{\beta}_1], \ldots, \alpha_n \mapsto sh[\vec{\alpha}_n, \vec{\beta}_n]]$
        let $\sim'$ be the least equivalence relation containing the pairs
              $\{(\alpha', \alpha'') \mid \alpha' \sim \alpha'' \wedge \{\alpha', \alpha''\} \cap \{\alpha_1, \cdots, \alpha_n\} = \emptyset\} \cup$
              $\{(\alpha_{0j}, \alpha_{ij}) \mid \vec{\alpha}_0 = \alpha_{01} \cdots \alpha_{0m}, \vec{\alpha}_i = \alpha_{i1} \cdots \alpha_{im}, 1 \leq i \leq n\}$
        in    if $\{\alpha_1, \cdots, \alpha_n\} \cap FV(t) = \emptyset$
              then $(R, \sim')$
              else return no answer
```

Figure 7: Forced matching for simple types.

We write $sh[\vec{t}, \vec{b}]$ for the type obtained by replacing all type holes with the relevant type in the list $\vec{t}$ and by replacing all behaviour holes with the relevant behaviour in the list $\vec{b}$; we shall dispense with a formal definition of this and we shall assume throughout that the length of $\vec{t}$ and $\vec{b}$ are adequate for the shape $sh$.

**Example 4.4** If $sh = ([\,] \times [\,])\ \texttt{com}\ [\,]$ then $sh[\vec{t}, \vec{b}] = (t_1 \times t_2)\ \texttt{com}\ b_1$ if and only if $\vec{t} = t_1 t_2$ and $\vec{b} = b_1$. □

The axioms (mr) and (ml) make use of the operation $\mathcal{M}$ defined in Figure 7 to force a type $t$ to match a type variable $\alpha$. The call $\mathcal{M}(\alpha, t, \sim) = (R, \sim')$ produces the substitution $R$ and modifies the equivalence relation $\sim$ (over the free variables of a constraint set $C'$) to another equivalence relation $\sim'$ (over the free variables of the constraint set $RC'$). In axioms (mr) and (ml) the newly produced substitution $R$ is composed with the previously produced substitutions. Also note that the "extended occurs check" in Figure 7 ensures that $Rt = t$.

Using Fact 3.5 it is straightforward to establish:

**Fact 4.5** Suppose $(S, C, \sim) \longrightarrow (S', C', \sim')$. Then there exists simple $R$ such that $S' = RS$ and such that $RC \rightharpoondown^* C'$. Moreover, if $S$ and $C$ are simple then also $S'$ and $C'$ are simple.

**Remark: type cycles become behaviour cycles.** To understand why $\mathcal{F}$ does not report failure in *more* cases than a "classical type checker", the following example is helpful. Consider the constraint set

$$C = \{\texttt{int} \rightarrow^{\alpha\ \mathrm{CHAN}} \ \texttt{int} \subseteq \alpha\}$$

which will not cause a classical type checker to fail since $\alpha$ is simply unified with $\texttt{int} \rightarrow \ \texttt{int}$. Now let us see how $\mathcal{F}$ behaves on this constraint, encoded as a set of *simple* constraints:

$$\{\texttt{int} \rightarrow^{\beta} \ \texttt{int} \subseteq \alpha,\ \{\alpha\ \mathrm{CHAN}\} \subseteq \beta\}.$$

Here case (mr) in Figure 6 is enabled, and consequentially a substitution which maps $\alpha$ into $\texttt{int} \rightarrow^{\beta'} \ \texttt{int}$ (with $\beta'$ new) is applied to the constraints. The resulting constraint set is

$$\{\texttt{int} \rightarrow^{\beta} \ \texttt{int} \subseteq \texttt{int} \rightarrow^{\beta'} \ \texttt{int},\ \{(\texttt{int} \rightarrow^{\beta'} \ \texttt{int})\ \mathrm{CHAN}\} \subseteq \beta\}$$

and after first applying case (dc) for ($\rightarrow$) and then applying case (dc) for ($\texttt{int}$) we end up with the constraint set

$$C' = \{\beta \subseteq \beta',\ \{(\texttt{int} \rightarrow^{\beta'} \ \texttt{int})\ \mathrm{CHAN}\} \subseteq \beta\}$$

which cannot be rewritten further. The set $C'$ is atomic and well-formed so Algorithm $\mathcal{F}$ succeeds on $C$. $\square$

## 4.1 Termination and Soundness of $\mathcal{F}$

Having completed the definition of $\mathcal{M}$, $\longrightarrow$ and $\mathcal{F}$ we can state:

**Lemma 4.6** $\mathcal{F}(C)$ always terminates (possibly with failure). If $\mathcal{F}(C) = (S', C')$ with $C$ simple then $S'$ and $C'$ are simple, well-formed and atomic. Moreover, $C'$ is determined from $S'\,C$ in the sense that $S'\,C \rightharpoonup^{*} C' \not\rightharpoonup$.

**Proof** To show that $\mathcal{F}$ always terminates it suffices to find a lexicographically defined well-founded order such that each rewrite decreases the measure according to the order. The first component of the measure of $(S, C, \sim)$ is the number of equivalence classes of $\sim$. The second component is a vector of numbers that for each index $i$ lists how many constraints in $C$ have size $i$; here the size of $(g_1 \subseteq g_2)$ may be taken to be the number of symbols occurring in it. Rewrites according to axioms (mr) or (ml) reduce the first component; all other rewrites keep the first component unchanged but decrease the second component.

It is easy to see that the other claims will follow provided we can show that if

$$(\mathrm{Id}, C, \mathrm{Eq}_C) \longrightarrow^{*} (S_n, C_n, \sim_n)$$

then $S_n\,C \rightharpoonup^* C_n$ and if $C$ is simple then $S_n$ and $C_n$ are simple. We do this by induction on the length of the derivation, where the base case as well as the part concerning simplicity (where we use Fact 4.5) is trivial. For the inductive step, suppose that

$$(\mathrm{Id}, C, \mathrm{Eq}_C) \longrightarrow^* (S_n, C_n, \sim_n) \longrightarrow (S_{n+1}, C_{n+1}, \sim_{n+1})$$

where the induction hypothesis ensures that $S_n\,C \rightharpoonup^* C_n$. By Fact 4.5 there exists $R$ such that $S_{n+1} = R\,S_n$ and such that $R\,C_n \rightharpoonup^* C_{n+1}$. As it is easy to see that the relation $\rightharpoonup$ is closed under substitution it holds that $R\,S_n\,C \rightharpoonup^* R\,C_n$, hence the claim. □

**Lemma 4.7** $\mathcal{F}$ *is sound*

If $\mathcal{F}(C) = (S', C')$ then $C' \vdash S'\,C$.

**Proof** By Lemma 4.6 we have $S'\,C \rightharpoonup^* C'$, which yields the claim due to Fact 4.1. □

**Remark.** By Fact 4.1 we know that $\rightharpoonup$ is confluent but this does not directly carry over to $\longrightarrow$ or $\mathcal{F}$: the constraint $\alpha_1 \subseteq \alpha_2$ may yield $([\alpha_1 \mapsto \alpha_0], \{\alpha_0 \subseteq \alpha_2\})$ as well as $([\alpha_2 \mapsto \alpha_0], \{\alpha_1 \subseteq \alpha_0\})$. However, Lemma 4.6 told us that $\mathcal{F}(C) = (S', C')$ ensures that $C'$ is determined from $S'\,C$, and we conjecture that $S'$ is determined (up to some notion of renaming) from $C$.

## 5 Algorithm $\mathcal{R}$

The purpose of algorithm $\mathcal{R}$ is to reduce the size of a constraint set which is already well-formed, atomic and simple. The techniques used are basically those of [9] and [2], adapted to our framework.

The transformation $\mathcal{R}$ may be described as a non-deterministic rewriting process, operating over triples of form $(C, t, b)$, and with respect to a fixed environment $A$. We then define $\mathcal{R}$ by:

$$\begin{array}{rl} \mathcal{R}(C, t, b, A) = \text{let} & (C', t', b') \text{ be given by} \\ & A \vdash (C, t, b) \longrightarrow^* (C', t', b') \not\longrightarrow \\ \text{in} & (C', t', b') \end{array}$$

The rewriting relation is defined by the axioms of Figure 8 and will be explained below (recall that $\dot\cup$ means disjoint union). To understand the axioms, it is

(redund) $A \vdash (C \dot{\cup} \{\gamma' \subseteq \gamma\}, t, b) \longrightarrow (C, t, b)$
provided $(\gamma' \Leftarrow^* \gamma) \in C$

(cycle) $A \vdash (C, t, b) \longrightarrow (S\,C, S\,t, S\,b)$
where $S = [\gamma \mapsto \gamma']$ with $\gamma \neq \gamma'$
provided $(\gamma \Leftarrow^* \gamma') \in C$ and $(\gamma' \Leftarrow^* \gamma) \in C$ and
provided $\gamma \notin FV(A)$

(shrink) $A \vdash (C \dot{\cup} \{\gamma' \subseteq \gamma\}, t, b) \longrightarrow (S\,C, S\,t, S\,b)$
where $S = [\gamma \mapsto \gamma']$ with $\gamma \neq \gamma'$
provided $\gamma \notin FV(RHS(C), A)$ and
provided $t$, $b$, and each element in $LHS(C)$ is monotonic in $\gamma$

(boost) $A \vdash (C \dot{\cup} \{\gamma \subseteq \gamma'\}, t, b) \longrightarrow (S\,C, S\,t, S\,b)$
where $S = [\gamma \mapsto \gamma']$ with $\gamma \neq \gamma'$
provided $\gamma \notin FV(A)$ and
provided $t$, $b$ and each element in $LHS(C)$ is anti-monotonic in $\gamma$

Figure 8: Eliminating constraints.

helpful to view the constraints as a directed graph where the nodes are type or behaviour variables or of form $\{t \text{ CHAN}\}$, and the arrows cannot have a node of form$\{t \text{ CHAN}\}$ as the source. To this end we define:

**Definition 5.1** We write $(\gamma \Leftarrow^* \gamma') \in C$ if there is a path from $\gamma'$ to $\gamma$: there exists $\gamma_0 \cdots \gamma_n$ $(n \geq 0)$ such that $\gamma_0 = \gamma$ and $\gamma_n = \gamma'$ and $(\gamma_i \subseteq \gamma_{i+1}) \in C$ for all $i \in \{0 \ldots n-1\}$.

Notice that $(\gamma \Leftarrow^* \gamma) \in C$ holds also if $\gamma \notin FV(C)$. From reflexivity and transitivity of $\subseteq$ we have:

**Fact 5.2** If $(\gamma \Leftarrow^* \gamma') \in C$ then also $C \vdash \gamma \subseteq \gamma'$.

We have a substitution result similar to Lemma 2.2:

**Fact 5.3** Let $S$ be a substitution mapping variables into variables, and suppose $(\gamma \Leftarrow^* \gamma') \in C$. Then also $(S\,\gamma \Leftarrow^* S\,\gamma') \in S\,C$.

We say that $C$ is cyclic if there exists $\gamma_1, \gamma_2 \in FV(C)$ with $\gamma_1 \neq \gamma_2$ such that $(\gamma_1 \Leftarrow^* \gamma_2) \in C$ and $(\gamma_2 \Leftarrow^* \gamma_1) \in C$.

We now explain the rules: (redund) removes constraints which are redundant due to the ordering $\subseteq$ being reflexive and transitive; applying this rule repeatedly is called "transitive reduction" in [9] and is essential for a compact representation of the constraints.

The remaining rules all replace some variable $\gamma$ by another variable $\gamma'$. However, unlike what is the case for $\mathcal{F}$ the substitution $[\gamma \mapsto \gamma']$ is *not* returned and is not applied to $A$; therefore we demand that $\gamma$ does not belong to $FV(A)$. This is not something that can easily be rectified: not all substitutions $S$ that solve $C$ can be written on the form $S'[\gamma \mapsto \gamma']$ and hence we could lose completeness if we relaxed our demand.

The rule (cycle) collapses cycles in the graph; due to the remark above a cycle which involves *two* elements of $FV(A)$ cannot be eliminated. (However, in [9] it holds that $\emptyset \vdash b_1 \equiv b_2$ implies $b_1 = b_2$ and hence cycle elimination can be part of the analogue of $\mathcal{F}$.)

The rule (shrink) expresses that a variable $\gamma$ can be replaced by its "immediate predecessor" $\gamma'$, and due to the ability to perform transitive reduction this can be strengthened to the requirement that $\gamma'$ is the "only predecessor" of $\gamma$, which can be formalised as the side condition $\gamma \notin FV(\mathit{RHS}(C))$ where $\mathit{RHS}(C) = \{\gamma \mid \exists g : (g \subseteq \gamma) \in C\}$. We can allow $\gamma$ to belong to $t$ and $b$ and $\mathit{LHS}(C)$, where $\mathit{LHS}(C) = \{g \mid \exists \gamma : (g \subseteq \gamma) \in C\}$, as long as we do not "lose instances", that is we must have that $S\,t \subseteq t$, $S\,b \subseteq b$, and $S\,g \subseteq g$ for each $g \in \mathit{LHS}(C)$. This will be the case provided $t$ and $b$ and each element of $\mathit{LHS}(C)$ are *monotonic* in $\gamma$, where for example $t = \alpha_1 \rightarrow^{\beta_1} \alpha_2 \rightarrow^{\beta_1} \alpha_1$ is monotonic in $\gamma$ for *all* $\gamma \notin \{\alpha_1, \alpha_2\}$. A more formal treatment of the concept of monotonicity will be given shortly, for now notice that if $\gamma \notin FV(g)$ or if $g = \gamma$ then $g$ is monotonic in $\gamma$.

The rule (boost) expresses that a variable $\gamma$ can be replaced by its "immediate successor" $\gamma'$, and due to the ability to perform transitive reduction this can be strengthened to the requirement that $\gamma'$ must be the "only successor" of $\gamma$. In addition we must demand that we do not "lose instances", that is we must have that $S\,t \subseteq t$, $S\,b \subseteq b$, and $S\,g \subseteq g$ for each $g \in \mathit{LHS}(C)$. This will be the case provided $t$ and $b$ and each element of $\mathit{LHS}(C)$ are *anti-monotonic* in $\gamma$, where for example $t = \alpha_1 \rightarrow^{\beta_1} \alpha_2 \rightarrow^{\beta_1} \alpha_1$ is anti-monotonic in $\gamma$ for *all* $\gamma \notin \{\alpha_1, \beta_1\}$. Notice that if each element of $\mathit{LHS}(C)$ is anti-monotonic in $\gamma$ then $\gamma'$ in fact is the only successor of $\gamma$.

## Monotonicity

**Definition 5.4** Given a constraint set $C$. We say that a substitution $S$ is increasing (respectively decreasing) wrt. $C$ if for all $\gamma$ we have $C \vdash \gamma \subseteq S\,\gamma$ (respectively $C \vdash S\,\gamma \subseteq \gamma$).

We say that a substitution $S$ increases (respectively decreases) $g$ wrt. $C$ whenever $C \vdash g \subseteq S\,g$ (respectively $C \vdash S\,g \subseteq g$).

We want to define the concepts of monotonicity and anti-monotonicity such that the following result holds:

**Lemma 5.5** Suppose that $g$ is monotonic in all $\gamma \in \mathit{Dom}(S)$; then if $S$ is increasing (respectively decreasing) wrt. $C$ then $S$ increases (respectively decreases) $g$ wrt. $C$.

Suppose that $g$ is anti-monotonic in all $\gamma \in \mathit{Dom}(S)$; then if $S$ is increasing (respectively decreasing) wrt. $C$ then $S$ decreases (respectively increases) $g$ wrt. $C$. □

To this end we make the following recursive definition of sets $\mathit{NN}(g)$ and $\mathit{NP}(g)$ (for "not negative" and "not positive"):

$$
\begin{aligned}
&\mathit{NN}(\gamma) = \{\gamma\} \text{ and } \mathit{NP}(\gamma) = \emptyset;\\
&\mathit{NN}(\mathtt{unit}) = \mathit{NN}(\mathtt{int}) = \mathit{NN}(\mathtt{bool}) = \mathit{NN}(\emptyset) = \emptyset;\\
&\mathit{NP}(\mathtt{unit}) = \mathit{NP}(\mathtt{int}) = \mathit{NP}(\mathtt{bool}) = \mathit{NP}(\emptyset) = \emptyset;\\
&\mathit{NN}(t_1 \to^b t_2) = \mathit{NP}(t_1) \cup \mathit{NN}(b) \cup \mathit{NN}(t_2);\\
&\mathit{NP}(t_1 \to^b t_2) = \mathit{NN}(t_1) \cup \mathit{NP}(b) \cup \mathit{NP}(t_2);\\
&\mathit{NN}(t_1 \times t_2) = \mathit{NN}(t_1) \cup \mathit{NN}(t_2) \text{ and } \mathit{NP}(t_1 \times t_2) = \mathit{NP}(t_1) \cup \mathit{NP}(t_2);\\
&\mathit{NN}(t\ \mathtt{list}) = \mathit{NN}(t) \text{ and } \mathit{NP}(t\ \mathtt{list}) = \mathit{NP}(t);\\
&\mathit{NN}(t\ \mathtt{chan}) = \mathit{NP}(t\ \mathtt{chan}) = \mathit{FV}(t);\\
&\mathit{NN}(t\ \mathtt{com}\ b) = \mathit{NN}(t) \cup \mathit{NN}(b) \text{ and } \mathit{NP}(t\ \mathtt{com}\ b) = \mathit{NP}(t) \cup \mathit{NP}(b);\\
&\mathit{NN}(\{t\ \text{CHAN}\}) = \mathit{NP}(\{t\ \text{CHAN}\}) = \mathit{FV}(t);\\
&\mathit{NN}(b_1 \cup b_2) = \mathit{NN}(b_1) \cup \mathit{NN}(b_2) \text{ and } \mathit{NP}(b_1 \cup b_2) = \mathit{NP}(b_1) \cup \mathit{NP}(b_2).
\end{aligned}
$$

We are now ready to define the concept "is monotonic in".

**Definition 5.6** We say that $g$ is monotonic in $\gamma$ if $\gamma \notin \mathit{NP}(g)$; and we say that $g$ is anti-monotonic in $\gamma$ if $\gamma \notin \mathit{NN}(g)$.

**Fact 5.7** For all types and behaviours $g$, it holds that $\mathit{NP}(g) \cup \mathit{NN}(g) = \mathit{FV}(g)$. (So if $g$ is monotonic as well as anti-monotonic in $\gamma$, then $\gamma \notin \mathit{FV}(g)$.)

Now we can prove Lemma 5.5:

**Proof** Induction on $g$, where there are two typical cases:

$g$ **is a variable:** The claims follow from the fact that if $g$ is anti-monotonic in all $\gamma \in \mathit{Dom}(S)$, then $g \notin \mathit{Dom}(S)$.

$g$ **is a function type** $t_1 \to^b t_2$**:** First consider the sub-case where $g$ is monotonic in all $\gamma \in \mathit{Dom}(S)$ and where $S$ is increasing wrt. $C$. Then $\gamma \in \mathit{Dom}(S)$ gives $\gamma \notin \mathit{NP}(t_1 \to^b t_2)$, and we infer that $\gamma \notin \mathit{NN}(t_1)$, $\gamma \notin \mathit{NP}(b)$, and $\gamma \notin \mathit{NP}(t_2)$, so that $t_1$ is anti-monotonic in $\gamma$ whereas $t_2$ and $b$ are monotonic in $\gamma$. We can thus apply the induction hypothesis to infer that $S$ decreases $t_1$ wrt. $C$ and that $S$ increases $t_2$ as well as $b$ wrt. $C$. But then it is straightforward that $S$ increases $g$ wrt. $C$.

The other sub-cases are similar. □

**Example 5.8** Let $C$ and $t$ be given by

$$C = \{\alpha_1 \subseteq \alpha_2\} \text{ and } t = \alpha_1 \to^\beta \alpha_2. \tag{1}$$

As $t$ is monotonic in $\alpha_2$, it is possible to apply (shrink) and get

$$C' = \emptyset \text{ and } t' = \alpha_1 \to^\beta \alpha_1. \tag{2}$$

The soundness and completeness of this transformation may informally be argued as follows: (1) "denotes" the set of types

$$\{t_1 \to^b t_2 \mid \emptyset \vdash t_1 \subseteq t_2\}$$

but this is also the set of types denoted by (2), due to the presence of subtyping.

Notice that since $t$ is anti-monotonic in $\alpha_1$, it is also possible to apply (boost) from (1) and arrive at

$$C' = \emptyset \text{ and } t' = \alpha_2 \to^\beta \alpha_2$$

which modulo renaming is equal to (2).

**Example 5.9** Let $C$ and $t$ be given by

$$C = \{\alpha_2 \subseteq \alpha_1\} \text{ and } t = \alpha_1 \to^\beta \alpha_2.$$

Then neither (shrink) nor (boost) is applicable, as $t$ is not monotonic in $\alpha_1$ nor anti-monotonic in $\alpha_2$.

## 5.1 Termination and Soundness of $\mathcal{R}$

**Lemma 5.10** Suppose that $C$, $t$ and $b$ are simple, well-formed and atomic; then $\mathcal{R}(C,t,b,A)$ always terminates successfully and if $\mathcal{R}(C,t,b,A) = (C',t',b')$ then $C'$, $t'$ and $b'$ are simple, well-formed and atomic.

**Proof** Termination is ensured since each rewriting step either decreases the number of constraints, or (as is the case for (cycle)) decreases the number of variables without increasing the number of constraints. Each rewriting step trivially preserves simplicity, well-formedness and atomicity. □

Turning to soundness, we first prove an auxiliary result about the rewriting relation:

**Lemma 5.11** Suppose $A \vdash (C,t,b) \longrightarrow (C',t',b')$. Then there exists $S$ such that $C' \vdash S\,C$, $t' = S\,t$, $b' = S\,b$, and $A = S\,A$.

**Proof** For (redund) we can use $S = \mathrm{Id}$ and the claim follows from Fact 5.2. For (cycle) the claim is trivial; and for (shrink) and (boost) the claim follows from the fact that with $(\gamma_1 \subseteq \gamma_2)$ the "discarded" constraint it holds that $(S\,\gamma_1 \subseteq S\,\gamma_2)$ is an instance of reflexivity. □

Using Lemma 2.2 and Lemma 2.3 we then get:

**Corollary 5.12** Suppose $A \vdash (C,t,b) \longrightarrow (C',t',b')$.
If $C, A \vdash e : t \,\&\, b$ then $C', A \vdash e : t' \,\&\, b'$.

By repeated application of this corollary we get the desired result:

**Lemma 5.13** Suppose that $\mathcal{R}(C,t,b,A) = (C',t',b')$.
If $C, A \vdash e : t \,\&\, b$ then $C', A \vdash e : t' \,\&\, b'$.

## 5.2 Results concerning Confluence and Determinism

We have the following result showing that no new paths are introduced in the graph:

**Lemma 5.14** Suppose $A \vdash (C',t',b') \longrightarrow (C'',t'',b'')$ and let $\gamma_1, \gamma_2 \in FV(C'')$. Then $(\gamma_1 \Leftarrow^* \gamma_2) \in C'$ holds iff $(\gamma_1 \Leftarrow^* \gamma_2) \in C''$ holds.

**Proof** See Appendix A. □

**Observation 5.15** Suppose $A \vdash (C,t,b) \longrightarrow (C',t',b')$ where the rule (cycle) is not applicable from the configuration $(C,t,b)$. Then the rule (cycle) is not applicable from the configuration $(C',t',b')$ either.

This suggests that an implementation could begin by collapsing all cycles once and for all, without having to worry about cycles again. On the other hand, it is not possible to perform transitive reduction in a separate phase as (redund) may become enabled after applying (shrink) or (boost): as an example consider the situation where $C$ contains the constraints

$$\gamma_0 \subseteq \gamma,\ \gamma \subseteq \gamma_1,$$
$$\gamma_0 \subseteq \gamma',\ \gamma' \subseteq \gamma_1$$

and (redund) is not applicable. By applying (shrink) with the substitution $[\gamma \mapsto \gamma_0]$ we end up with the constraints

$$\gamma_0 \subseteq \gamma_1,\ \gamma_0 \subseteq \gamma',\ \gamma' \subseteq \gamma_1$$

of which the former can be eliminated by (redund).

Concerning confluency, one would like to show a "diamond property" but this cannot be done in the presence of cycles in the constraint set (especially if these contain multiple elements of $FV(A)$): as an example consider the constraints

$$\gamma_0 \subseteq \gamma,\ \gamma_0 \subseteq \gamma',\ \gamma \subseteq \gamma',\ \gamma' \subseteq \gamma$$

with $\gamma, \gamma' \in FV(A)$; here we can apply (redund) to eliminate either the first or the second constraint but then we are stuck as (cycle) is not applicable and therefore we cannot complete the diamond. As another example, consider the case where we have a cycle containing $\gamma_0$, $\gamma_1$ and $\gamma_2$ with $\gamma_0, \gamma_1 \in FV(A)$. Then we can apply (cycle) to map $\gamma_2$ into either $\gamma_0$ or $\gamma_1$ but then we are stuck and the graphs will be different (due to the arrows to or from $\gamma_2$) unless we devise some notion of graph equivalence.

On the other hand, we have the following result:

**Proposition 5.16** Suppose that

$$A \vdash (C, t, b) \longrightarrow (C_1, t_1, b_1) \text{ and}$$
$$A \vdash (C, t, b) \longrightarrow (C_2, t_2, b_2)$$

where $C$ is *acyclic* as well as simple, atomic and well-formed. Then there exists $(C_1', t_1', b_1')$ and $(C_2', t_2', b_2')$, which are equal up to renaming, such that

$$A \vdash (C_1, t_1, b_1) \longrightarrow^{\leq 1} (C_1', t_1', b_1') \text{ and}$$
$$A \vdash (C_2, t_2, b_2) \longrightarrow^{\leq 1} (C_2', t_2', b_2').$$

**Proof** See Appendix A. □

(lub-exists) $A \vdash (C \,\dot{\cup}\, \{b_1 \subseteq \beta\} \,\dot{\cup}\, \cdots \,\dot{\cup}\, \{b_n \subseteq \beta\}, t, b) \longrightarrow (C, t, b)$
provided $\beta \notin FV(C, t, b, A)$ and $\beta \notin FV(b_1, \cdots, b_n)$

(shrink-chan) $A \vdash (C \,\dot{\cup}\, \{\{t' \text{ CHAN}\} \subseteq \beta\}, t, b) \longrightarrow (S\,C, S\,t, S\,b)$
where $S = [\beta \mapsto \{t' \text{ CHAN}\}]$
provided $\beta \notin FV(RHS(C), A, t')$ and
provided $t$, $b$, and each element in $LHS(C)$ is monotonic in $\beta$ and
provided that $S\,t$ and $S\,b$ and $S\,C$ are simple

(shrink-empty) $A \vdash (C, t, b) \longrightarrow (C', S\,t, S\,b)$
with $C' = \{(g_1 \subseteq g_2) \in S\,C \mid g_1 \neq \emptyset\}$
where $S = [\beta \mapsto \emptyset]$ with $\beta \in FV(C, t, b)$
provided $\beta \notin FV(RHS(C), A)$ and
provided $t$, $b$, and each element in $LHS(C)$ is monotonic in $\beta$ and
provided that $S\,t$ and $S\,b$ and $C'$ are simple

Figure 9: Additional simplifications.

## 5.3 Extensions of $\mathcal{R}$

In addition to the rewritings presented in Figure 8 one might introduce several other rules, some of which are listed in Figure 9.

The rule (lub-exists) allows us to dispense with constraints which state that some behaviours have an upper bound, as long as this upper bound does not occur elsewhere. Notice that a similar rule for types would be invalid, since two types do not necessarily possess an upper bound.

The rules (shrink-chan) and (shrink-empty) extend (shrink) in that they replace a variable $\gamma$ by its "immediate predecessor" $g$ even if $g$ is not a variable: for (shrink-chan) $g$ is a behaviour $\{t' \text{ CHAN}\}$ (where an "occur check" has to be performed), and for (shrink-empty) it is $\emptyset$ (which is a "trivial predecessor").

For the rules (shrink-chan) and (shrink-empty), we must preserve the property of being simple and we have explicit clauses for ensuring this; we also must preserve atomicity and therefore rule (shrink-empty) discards all constraints with $\emptyset$ on the left hand side.

## Termination and soundness

Adding the rules in Figure 9 preserves termination and soundness, as it is easy to see that Lemma 5.10 and Lemma 5.11 still hold: for (shrink-chan) we employ the side condition $\beta \notin FV(t')$; for (shrink-empty) we employ that $\emptyset$ is the least behaviour; for (lub-exists) we use $S = [\beta \mapsto b_1 \cup \cdots \cup b_n]$ and then employ that $\cup$ is an upper bound operator, together with the side condition $\beta \notin FV(b_1, \cdots, b_n)$.

Notice, however, that it no longer holds in general that the substitution $S$ used in Lemma 5.11 is simple; so if we were to extend $\mathcal{R}$ with the rules in Figure 9 we would lose the property that the inference tree "constructed by" the inference algorithm is "simple".

## 6 Experimental Results

In [8] we considered the program

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

which demonstrated the power of our inference system relative to other approaches. Analysing this program with $\mathcal{R}$ as described in Figure 8, our prototype implementation produces 4 type constraints and 7 behaviour constraints. The resulting simple type is

$$((\alpha_{44} \xrightarrow{\beta_{20}} \alpha_{45}) \xrightarrow{\beta_{34}} (\alpha_{54} \xrightarrow{\beta_{31}} \alpha_{54}))$$

the resulting behaviour is $\emptyset$ and the resulting type constraints are

$$\alpha_{44} \subseteq \alpha_{45},\ \alpha_{54} \subseteq \alpha_{49},\ \alpha_{58} \subseteq \alpha_{54},\ \alpha_{54} \subseteq \alpha_{59}$$

and the resulting behaviour constraints are

$$\begin{array}{l} \beta_{20} \subseteq \beta_{23},\ \{\alpha_7\, \mathrm{CHAN}\} \subseteq \beta_{23}, \\ \beta_{20} \subseteq \beta_{25},\ \{(\alpha_{58} \xrightarrow{\beta_{33}} \alpha_{59})\, \mathrm{CHAN}\} \subseteq \beta_{25}, \\ \beta_{20} \subseteq \beta_{27},\ \{\alpha_{49}\, \mathrm{CHAN}\} \subseteq \beta_{27}, \\ \beta_{31} \subseteq \beta_{33}. \end{array}$$

**Remark.** Analysing the program above with a version of $\mathcal{R}$ which uses only (redund) and (cycle) but not (shrink) or (boost), our implementation produces 71 type constraints and 88 behaviour constraints. This shows that it is essential to use a non-trivial version of $\mathcal{R}$ in order to get readable output. Alternatively, (shrink) and (boost) could be applied only in the top-level call to $\mathcal{W}$; then the implementation produces a result isomorphic to the one above (4 type constraints and 7 behaviour constraints), but is much slower (due to the need to carry around a large set of constraints).

**Additional simplifications.** This not quite as informative as we might wish, which suggests that $\mathcal{R}$ should be extended with the rules in Figure 9: by applying (lub-exists) repeatedly we can eliminate 6 of the behaviour constraints such that the remaining type and behaviour constraints are

$$\alpha_{44} \subseteq \alpha_{45},\ \alpha_{54} \subseteq \alpha_{49},\ \alpha_{58} \subseteq \alpha_{54},\ \alpha_{54} \subseteq \alpha_{59},\ \beta_{31} \subseteq \beta_{33}$$

and this makes it possible to shrink $\beta_{33}$, $\alpha_{49}$, and $\alpha_{59}$ and to boost $\alpha_{58}$ such that we end up with one constraint only:

$$((\alpha_{44} \overset{\beta_{20}}{\to} \alpha_{45}) \overset{\beta_{34}}{\to} (\alpha_{54} \overset{\beta_{31}}{\to} \alpha_{54}))$$
$$\text{where } \alpha_{44} \subseteq \alpha_{45}$$

This is small enough to be manageable and is actually more precise than the (essentially simple) type

$$(\alpha \to^{\beta} \alpha) \to^{\emptyset} (\alpha' \to^{\emptyset} \alpha')$$

(and no constraints) that is perhaps closer to what the programmer might have expected.

## 7 Syntactic Soundness of Algorithm $\mathcal{W}$

A main technical property of algorithm $\mathcal{W}$ is that it always terminates:

**Lemma 7.1** If $A$ is simple then $\mathcal{W}(A, e)$ always terminates (possibly with failure); if $\mathcal{W}(A, e) = (S, t, b, C)$ then $S$, $t$, $b$, and $C$ are simple, well-formed and atomic.

**Proof** This result is proved by structural induction in $e$ with a similar result for $\mathcal{W}'$ except that $\mathcal{W}'(A, e) = (S, t, b, C)$ neither ensures that $C$ is well-formed nor atomic. For $\mathcal{F}$ and $\mathcal{R}$ we employ Lemma 4.6 and Lemma 5.10; for constants we employ Fact 3.4; and throughout we employ Fact 3.5. □

Note that if the expression $e$ only mentions identifiers in the domain of $A$ (as when $e$ is closed), and that if $e$ only mentions constants for which TypeOf is defined, then the only possible form for failure is due to $\mathcal{F}$. We conjecture that then also ML typing would have failed (cf. the discussion in Section 3).

As a final preparation for establishing soundness of algorithm $\mathcal{W}$ we establish a result about our formula for generalisation.

**Lemma 7.2** Let $C$ be well-formed; then $C, A \vdash e : t \,\&\, b$ holds if and only if $C, A \vdash e : \mathit{GEN}(A,b)(C,t) \,\&\, b$.

**Proof** See Appendix A. □

**Theorem 7.3** If $\mathcal{W}(A,e) = (S,t,b,C)$ with $A$ simple then $C, S\,A \vdash e : t \,\&\, b$.

**Proof** The result is shown by induction in $e$ with a similar result for $\mathcal{W}'$. See Appendix A for the details. □

## 8 Solvability of the Constraints Generated

Typability of an expression $e$ might be taken to mean

$$\exists t, b : \emptyset, [\,] \vdash e : t \,\&\, b$$

where $\emptyset$ is the empty constraint set and $[\,]$ is the empty assumption list. To check for typability it is natural to perform a call $\mathcal{W}([\,],e)$ and to determine whether or not the call

$$\mathcal{W}([\,],e) \text{ terminates successfully} \qquad (1)$$

rather than with failure (recalling that by Lemma 7.1 the call $\mathcal{W}([\,],e)$ must terminate). We conjecture a completeness property showing that (1) is a *necessary* condition for (certain kinds of) typability; here we shall be content with asking whether (1) is a *sufficient* condition for typability.

If $\mathcal{W}([\,],e)$ terminates successfully producing $(S,t,b,C)$ it follows from the Soundness Theorem 7.3 together with Lemma 7.1 that $C, [\,] \vdash e : t \,\&\, b$ with $C$ simple, well-formed and atomic; but to achieve typability we must achieve an empty constraint set. Due to the substitution and entailment lemmas (2.2 and 2.3) it will suffice to find a substitution $S'$ such that $\emptyset \vdash S'\,C$ for then we have a judgement of the desired form: $\emptyset, [\,] \vdash e : S'\,t \,\&\, S'\,b$.

Our goal thus is:

$$\text{Given simple, well-formed and atomic } C\text{; find } S' \text{ such that } \emptyset \vdash S'\,C. \qquad (2)$$

This is a kind of simplification process and as this paper does not address completeness issues we shall not be concerned with principality (that $\emptyset \vdash S''\,C$ implies that $S''$ can be written as $S'''\,S'$).

We shall construct the $S'$ mentioned in (2) by the formula $S' = S'_3\,S'_2\,S'_1$ where $S'_1$ solves the type constraints of form $(\alpha_1 \subseteq \alpha_2)$, where $S'_2$ solves the behaviour constraints of form $(\beta_1 \subseteq \beta_2)$, and $S'_3$ solves the behaviour constraints of form $(\{t \text{ CHAN}\} \subseteq \beta_2)$. By simplicity, well-formedness, and atomicity of $C$ this takes care of all constraints of $C$ (provided we demand that $S'_1$ and $S'_2$ preserve simplicity, well-formedness and atomicity).

A crude approach to defining $S'_1$ is to select a unique type variable $\alpha_*$ and let $S'_1$ map all type variables of $FV(C)$ to $\alpha_*$. Clearly this solves all type constraints of $C$ in the sense that all type constraints of $S'_1\,C$ are of the form $(\alpha_* \subseteq \alpha_*)$ and hence instances of the axiom of reflexivity. (A less crude approach would be to consider each $(\alpha_1 \subseteq \alpha_2)$ of $C$ in turn and perform a most general unification of $\alpha_1$ with $\alpha_2$.)

A crude approach to defining $S'_2$ is to select a unique behaviour variable $\beta_*$ and to let $S'_2$ map all behaviour variables of $FV(S'_1\,C)$ to $\beta_*$. Clearly this solves all behaviour constraints in $C$ that were of the form $(\beta_1 \subseteq \beta_2)$ since in $S'_2\,S'_1\,C$ they appear as $(\beta_* \subseteq \beta_*)$. (A less crude approach would be to adopt the ideas of canonical solution from [10] but this is best combined with the construction of $S'_3$ below.)

The remaining non-trivial constraints in $S'_2\,S'_1\,C$ are $\{t_1 \text{ CHAN}\} \subseteq \beta_*, \cdots, \{t_n \text{ CHAN}\} \subseteq \beta_*$ for $n \geq 0$. If $\beta_*$ does not occur in any of $t_1, \cdots, t_n$ we could follow [10] and define $S'_3$ by letting it map $\beta_*$ to $\beta_* \cup \{t_1 \text{ CHAN}\} \cup \cdots \cup \{t_n \text{ CHAN}\}$ and perhaps even dispense with the "$\beta_* \cup$ ". This situation corresponds to the scenario in [10] where the type inference algorithm enforces that $\beta_*$ does not occur in $t_1, \cdots, t_n$ by terminating with failure if the condition is not met. Intuitively, failure to meet the condition means that the communication capabilities are used to code up recursion in "much the same way" that the $Y$ combinatior can be encoded in the $\lambda$-calculus with recursive types (or in the untyped $\lambda$-calculus). However, we shall take the view that it is too demanding to always forbid such use of the communication capabilities and thus depart from [10].

It is important to note that a simple solution could be found if we *changed the representation* of constraints to record their free variables only: then $\{\{t_1 \text{ CHAN}\} \subseteq \beta_*, \cdots, \{t_n \text{ CHAN}\} \subseteq \beta_*\}$ is replaced by $\{(\gamma \subseteq \beta_*) \mid \gamma \in \bigcup_i FV(t_i)\}$. Even if $\beta_*$ occurs in one of $t_1, \cdots, t_n$ one could still let $S'_3$ map $\beta_*$ to $\beta_* \cup \gamma_1 \cup \cdots \cup \gamma_m$ where $\{\gamma_1, \cdots, \gamma_m\} = FV(t_1, \cdots, t_n)$ and we could obtain a solution due to the axioms for $\cup$ in Figure 2. In many ways this would seem a sensible solution in that the actual structure of the type is of only minor importance.

Motivated by the goals of [6] of eventually incorporating more causal information also for behaviours, we shall favour another solution. This involves adding a new behaviour of form $REC\beta.b$. Formally we extend the syntax as in

$$b ::= \cdots \mid REC\beta.b$$

and extend the axiomatisation of Figure 2 with the axiom scheme

$$C \vdash (REC\beta.b) \equiv b[(REC\beta.b)/\beta]$$

For a constraint set $C$ to be simple we require that there are no occurrences of $REC$ in it. With this new form of behaviour we can define $S_3'$ by mapping $\beta_*$ to $REC\beta_*.(\beta_* \cup \{t_1 \text{ CHAN}\} \cup \cdots \cup \{t_n \text{ CHAN}\})$. We then have $\emptyset \vdash S_3' \, S_2' \, S_1' \, C$ as desired.

## 9 Conclusion

We have developed an inference algorithm for a previously developed annotated type and effect system that integrates polymorphism, subtyping and effects [8]. Although the development was performed for a fragment of Concurrent ML we believe it equally possible for Standard ML with references. The algorithm $\mathcal{W}$ involves the syntactically defined $\mathcal{W}'$, and the algorithm $\mathcal{F}$ for obtaining constraints that are well-formed; an optional component, algorithm $\mathcal{R}$ for reducing the size of constraint sets, is pragmatically very useful in reducing constraint sets to a manageable size, as is illustrated in our prototype implementation. In this paper we showed the syntactic soundness of these algorithms and the issue of completeness seems promising.

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

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

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

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

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

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

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

[9] 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.

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

[11] J. P. Talpin and P. Jouvelot. Polymorphic Type, Region and Effect Inference. *Journal of Functional Programming*, 2(3), pages 245–271, 1992.

# A Details of Proofs

## Algorithm $\mathcal{R}$

**Lemma 5.14** Suppose $A \vdash (C', t', b') \longrightarrow (C'', t'', b'')$ and let $\gamma_1, \gamma_2 \in FV(C'')$. Then $(\gamma_1 \Leftarrow^* \gamma_2) \in C'$ holds iff $(\gamma_1 \Leftarrow^* \gamma_2) \in C''$ holds.

**Proof** (We use the terminology from the relevant clauses in Figure 8, which does not conflict with the one used in the formulation of the lemma). For (redund) this is a straight-forward consequence of the assumptions. For (cycle), (shrink) and (boost) the "only if"-part follows from Fact 5.3: if $(\gamma_1 \Leftarrow^* \gamma_2) \in C'$ then $(S\,\gamma_1 \Leftarrow^* S\,\gamma_2) \in S\,C'$ and as $\gamma_1, \gamma_2 \notin \mathit{Dom}(S)$ this amounts to $(\gamma_1 \Leftarrow^* \gamma_2) \in S\,C'$ which is clearly equivalent to $(\gamma_1 \Leftarrow^* \gamma_2) \in C''$.

We are left with proving the "if"-part for (cycle), (shrink) and (boost); to do so it suffices to show that

$$(\gamma_1' \subseteq \gamma_2') \in C'' \text{ implies } (\gamma_1' \Leftarrow^* \gamma_2') \in C'.$$

As $C'' = S\,C$ we can assume that there exists $(\gamma_1 \subseteq \gamma_2) \in C$ such that $\gamma_1' = S\,\gamma_1$ and $\gamma_2' = S\,\gamma_2$; then (since $C \subseteq C'$) our task can be accomplished by showing that

$$(S\,\gamma_1 \Leftarrow^* \gamma_1) \in C' \text{ and } (\gamma_2 \Leftarrow^* S\,\gamma_2) \in C'.$$

This is trivial except if $\gamma_1 = \gamma$ or $\gamma_2 = \gamma$. The former is impossible in the case (boost) (as $\mathit{LHS}(C)$ is anti-monotonic in $\gamma$) and otherwise the claim follows from the assumptions; the latter is impossible in the case (shrink) (as $\gamma \notin \mathit{RHS}(C)$) and otherwise the claim follows from the assumptions. $\square$

**Proposition 5.16** Suppose that

$$A \vdash (C, t, b) \longrightarrow (C_1, t_1, b_1) \text{ and}$$
$$A \vdash (C, t, b) \longrightarrow (C_2, t_2, b_2)$$

where $C$ is *acyclic* as well as simple, atomic and well-formed. Then there exists $(C_1', t_1', b_1')$ and $(C_2', t_2', b_2')$, which are equal up to renaming, such that

$$A \vdash (C_1, t_1, b_1) \longrightarrow^{\leq 1} (C_1', t_1', b_1') \text{ and}$$
$$A \vdash (C_2, t_2, b_2) \longrightarrow^{\leq 1} (C_2', t_2', b_2').$$

**Proof** As (cycle) is not applicable, each of the two rewriting steps in the assumption can be of three kinds yielding six different combinations:

**(redund) and (redund)** eliminating $(\gamma_1' \subseteq \gamma_1)$ and $(\gamma_2' \subseteq \gamma_2)$ where we can assume that either $\gamma_1' \neq \gamma_2'$ or $\gamma_1 \neq \gamma_2$ as otherwise the claim is trivial. The situation thus is

$$A \vdash (C \dot{\cup} \{\gamma_1' \subseteq \gamma_1\} \dot{\cup} \{\gamma_2' \subseteq \gamma_2\}, t, b) \longrightarrow (C \dot{\cup} \{\gamma_2' \subseteq \gamma_2\}, t, b)$$
$$A \vdash (C \dot{\cup} \{\gamma_1' \subseteq \gamma_1\} \dot{\cup} \{\gamma_2' \subseteq \gamma_2\}, t, b) \longrightarrow (C \dot{\cup} \{\gamma_1' \subseteq \gamma_1\}, t, b)$$

where

$$(\gamma_1' \Leftarrow^* \gamma_1) \in C \dot{\cup} \{\gamma_2' \subseteq \gamma_2\} \text{ and} \tag{1}$$
$$(\gamma_2' \Leftarrow^* \gamma_2) \in C \dot{\cup} \{\gamma_1' \subseteq \gamma_1\}. \tag{2}$$

It will suffice to show that

$$\text{either } (\gamma_1' \Leftarrow^* \gamma_1) \in C \text{ or } (\gamma_2' \Leftarrow^* \gamma_2) \in C \tag{3}$$

for if e.g. $(\gamma_1' \Leftarrow^* \gamma_1) \in C$ holds then by (2) also $(\gamma_2' \Leftarrow^* \gamma_2) \in C$ holds and we can apply (redund) twice to complete the diamond.

For the sake of arriving at a contradiction we now assume that (3) does not hold. Using (1) and (2) we see that the situation is that

$$(\gamma_1' \Leftarrow^* \gamma_2') \in C \text{ and } (\gamma_2 \Leftarrow^* \gamma_1) \in C \text{ and}$$
$$(\gamma_2' \Leftarrow^* \gamma_1') \in C \text{ and } (\gamma_1 \Leftarrow^* \gamma_2) \in C$$

and this conflicts with the assumption about the graph being cycle-free.

**(redund) and (shrink)** eliminating $(\gamma_1' \subseteq \gamma_1)$ and shrinking $\gamma_2$ into $\gamma_2'$ (with $\gamma_2' \neq \gamma_2$). First notice that it cannot be the case that $(\gamma_1' \subseteq \gamma_1) = (\gamma_2' \subseteq \gamma_2)$, for then (with $C$ the remaining constraints) we would have $(\gamma_1' \Leftarrow^* \gamma_1) \in C$ as well as $\gamma_2 \notin \mathit{RHS}(C)$. The situation thus is

$$A \vdash (C \dot{\cup} \{\gamma_1' \subseteq \gamma_1\} \dot{\cup} \{\gamma_2' \subseteq \gamma_2\}, t, b) \longrightarrow (C \dot{\cup} \{\gamma_2' \subseteq \gamma_2\}, t, b)$$
$$A \vdash (C \dot{\cup} \{\gamma_1' \subseteq \gamma_1\} \dot{\cup} \{\gamma_2' \subseteq \gamma_2\}, t, b) \longrightarrow (S\,C \cup \{S\,\gamma_1' \subseteq S\,\gamma_1\}, S\,t, S\,b)$$

where $S = [\gamma_2 \mapsto \gamma_2']$ and where

$$(\gamma_1' \Leftarrow^* \gamma_1) \in C \cup \{\gamma_2' \subseteq \gamma_2\} \text{ and}$$
$$\gamma_2 \notin \mathit{FV}(\mathit{RHS}(C), A) \text{ and } \gamma_2 \neq \gamma_1 \text{ and } t,\, b,\, \mathit{LHS}(C) \text{ is monotonic in } \gamma_2.$$

Applying Fact 5.3 we get $(S\,\gamma_1' \Leftarrow^* S\,\gamma_1) \in S\,C$ which shows that

$$A \vdash (S\,C \,\cup\, \{S\,\gamma_1' \subseteq S\,\gamma_1\}, S\,t, S\,b) \longrightarrow^{\leq 1} (S\,C, S\,t, S\,b)$$

(if $(S\,\gamma_1' \subseteq S\,\gamma_1) \in S\,C$ we have "$=$" otherwise "$\longrightarrow$"); it is also easy to see that the conditions are fulfilled for applying (shrink) to get

$$A \vdash (C \,\dot{\cup}\, \{\gamma_2' \subseteq \gamma_2\}, t, b) \longrightarrow (S\,C, S\,t, S\,b)$$

thus completing the diamond.

**(redund) and (boost)** eliminating $(\gamma_1 \subseteq \gamma_1')$ and boosting $\gamma_2$ into $\gamma_2'$ (with $\gamma_2' \neq \gamma_2$). First notice that it cannot be the case that $(\gamma_1 \subseteq \gamma_1') = (\gamma_2 \subseteq \gamma_2')$, for then (with $C$ the remaining constraints) we would have $(\gamma_1 \Leftarrow^* \gamma_1') \in C$ showing that $\gamma_1 \in \mathit{LHS}(C)$, whereas a side condition for (boost) is that each element in $\mathit{LHS}(C)$ is anti-monotonic in $\gamma_2$.

Now we can proceed as in the case (redund),(shrink).

**(shrink) and (shrink)** shrinking $\gamma_1$ into $\gamma_1'$ and shrinking $\gamma_2$ into $\gamma_2'$ where we can assume that either $\gamma_1' \neq \gamma_2'$ or $\gamma_1 \neq \gamma_2$ as otherwise the claim is trivial. The situation thus is

$$\begin{array}{l} A \vdash (C \,\dot{\cup}\, \{\gamma_1' \subseteq \gamma_1\} \,\dot{\cup}\, \{\gamma_2' \subseteq \gamma_2\}, t, b) \longrightarrow (S_1\,C \,\cup\, \{S_1\,\gamma_2' \subseteq S_1\,\gamma_2\}, S_1\,t, S_1\,b) \\ A \vdash (C \,\dot{\cup}\, \{\gamma_1' \subseteq \gamma_1\} \,\dot{\cup}\, \{\gamma_2' \subseteq \gamma_2\}, t, b) \longrightarrow (S_2\,C \,\cup\, \{S_2\,\gamma_1' \subseteq S_2\,\gamma_1\}, S_2\,t, S_2\,b) \end{array}$$

where $S_1 = [\gamma_1 \mapsto \gamma_1']$ and $S_2 = [\gamma_2 \mapsto \gamma_2']$. Due to the side conditions for (shrink) we have $\gamma_1 \neq \gamma_2$ and $\gamma_1, \gamma_2 \notin \mathit{RHS}(C)$ implying $\gamma_1, \gamma_2 \notin \mathit{RHS}(S_1\,C)$ and $\gamma_1, \gamma_2 \notin \mathit{RHS}(S_2\,C)$, thus the "$\cup$" on the right hand sides is really "$\dot{\cup}$". Our goal then is to find $S_1'$ and $S_2'$ such that

$$\begin{array}{l} S_2'\,S_1 = S_1'\,S_2 \text{ and} \\ A \vdash (S_1\,C \,\dot{\cup}\, \{S_1\,\gamma_2' \subseteq \gamma_2\}, S_1\,t, S_1\,b) \longrightarrow (S_2'\,S_1\,C, S_2'\,S_1\,t, S_2'\,S_1\,b) \text{ and} \\ A \vdash (S_2\,C \,\dot{\cup}\, \{S_2\,\gamma_1' \subseteq \gamma_1\}, S_2\,t, S_2\,b) \longrightarrow (S_1'\,S_2\,C, S_1'\,S_2\,t, S_1'\,S_2\,b). \end{array}$$

We naturally define $S_1' = [\gamma_1 \mapsto S_2\,\gamma_1']$ and $S_2' = [\gamma_2 \mapsto S_1\,\gamma_2']$ with the purpose of using (shrink), and our proof obligations are:

$$S_2'\,S_1 = S_1'\,S_2; \tag{4}$$
$$S_2\,\gamma_1' \neq \gamma_1 \text{ and } S_1\,\gamma_2' \neq \gamma_2; \tag{5}$$
$$S_2\,t, S_2\,b, \mathit{LHS}(S_2\,C) \text{ is monotonic in } \gamma_1; \tag{6}$$
$$S_1\,t, S_1\,b, \mathit{LHS}(S_1\,C) \text{ is monotonic in } \gamma_2. \tag{7}$$

Here (4) and (5) amounts to proving that

$$S_2'\,\gamma_1' = S_2\,\gamma_1' \text{ and } S_1\,\gamma_2' = S_1'\,\gamma_2' \text{ and } S_2\,\gamma_1' \neq \gamma_1 \text{ and } S_1\,\gamma_2' \neq \gamma_2 \tag{8}$$

which is trivial if $\gamma_1' \neq \gamma_2$ and $\gamma_2' \neq \gamma_1$. If e.g. $\gamma_1' = \gamma_2$ then we from our assumption about the graph being cycle-free infer that $\gamma_2' \neq \gamma_1$ from which (8) easily follows.

The claims (6) and (7) are easy consequences of the fact that $t$, $b$ and $LHS(C)$ are monotonic in $\gamma_1$ as well as in $\gamma_2$: for then we for instance have $\{\gamma_1, \gamma_2\} \cap NP(t) = \emptyset$ and hence $NP(S_1\,t) = NP(S_2\,t) = NP(t)$.

**(boost) and (boost)** where we proceed, *mutatis mutandis*, as in the case (shrink),(shrink).

**(shrink) and (boost)** shrinking $\gamma_1$ into $\gamma_1'$ and boosting $\gamma_2$ into $\gamma_2'$. Let $S_1 = [\gamma_1 \mapsto \gamma_1']$ and $S_2 = [\gamma_2 \mapsto \gamma_2']$. Four cases:

$\underline{\gamma_1 = \gamma_2}$ (to be denoted $\gamma$). Then our assumption about the graph being cycle-free tells us that $\gamma_1' \neq \gamma_2'$, and the situation is

$$A \vdash (C \dot\cup \{\gamma_1' \subseteq \gamma\} \dot\cup \{\gamma \subseteq \gamma_2'\}, t, b) \longrightarrow (S_1\,C \,\cup\, \{\gamma_1' \subseteq \gamma_2'\}, S_1\,t, S_1\,b) \tag{9}$$
$$A \vdash (C \dot\cup \{\gamma_1' \subseteq \gamma\} \dot\cup \{\gamma \subseteq \gamma_2'\}, t, b) \longrightarrow (S_2\,C \,\cup\, \{\gamma_1' \subseteq \gamma_2'\}, S_2\,t, S_2\,b) \tag{10}$$

where (according to the side conditions for (shrink) and (boost)) it holds that $\gamma \notin RHS(C)$ and that $t$, $b$ and each element in $LHS(C)$ is monotonic as well as anti-monotonic in $\gamma$. By Fact 5.7 and using that $C$ is well-formed we infer that $\gamma \notin FV(C, t, b)$, thus the right hand sides of (9) and (10) are identical.

$\underline{\gamma_1 = \gamma_2'}$. By the side condition for (shrink) we then have $\gamma_2 = \gamma_1'$. The situation thus is

$$A \vdash (C \dot\cup \{\gamma_2 \subseteq \gamma_1\}, t, b) \longrightarrow (S_1\,C, S_1\,t, S_1\,b)$$
$$A \vdash (C \dot\cup \{\gamma_2 \subseteq \gamma_1\}, t, b) \longrightarrow (S_2\,C, S_2\,t, S_2\,b)$$

where the right hand sides are equal modulo renaming.

$\underline{\gamma_2 = \gamma_1'}$. By the side condition for (boost) we then have $\gamma_1 = \gamma_2'$ so we can proceed as in the previous case.

$\underline{\gamma_1 \notin \{\gamma_2, \gamma_2', \gamma_1'\} \text{ and } \gamma_2 \notin \{\gamma_1, \gamma_1', \gamma_2'\}}$ will hold in the remaining case. The situation thus is

$$A \vdash (C \dot\cup \{\gamma_1' \subseteq \gamma_1\} \dot\cup \{\gamma_2 \subseteq \gamma_2'\}, t, b) \longrightarrow (S_1\,C \,\cup\, \{\gamma_2 \subseteq \gamma_2'\}, S_1\,t, S_1\,b)$$
$$A \vdash (C \dot\cup \{\gamma_1' \subseteq \gamma_1\} \dot\cup \{\gamma_2 \subseteq \gamma_2'\}, t, b) \longrightarrow (S_2\,C \,\cup\, \{\gamma_1' \subseteq \gamma_1\}, S_2\,t, S_2\,b)$$

where $\gamma_1 \notin FV(RHS(C), A)$, where $t$ and $b$ and each element in $LHS(C)$ is monotonic in $\gamma_1$, where $\gamma_2 \notin FV(A)$, and where $t$ and $b$ and each element in $LHS(C)$ is anti-monotonic in $\gamma_2$.

As $\gamma_1 \neq \gamma_2'$ it is easy to see that $\gamma_1 \notin FV(RHS(S_2\,C), A)$ and that $S_2\,t$, $S_2\,b$ and $LHS(S_2\,C)$ is monotonic in $\gamma_1$; and as $\gamma_2 \neq \gamma_1'$ it is easy to see that $S_1\,t$, $S_1\,b$ and $LHS(S_1\,C)$ is anti-monotonic in $\gamma_2$. Hence we can apply (boost) and (shrink) to get

$$A \vdash (S_1\,C \,\dot{\cup}\, \{\gamma_2 \subseteq \gamma_2'\}, S_1\,t, S_1\,b) \longrightarrow (S_2\,S_1\,C, S_2\,S_1\,t, S_2\,S_1\,b)$$
$$A \vdash (S_2\,C \,\dot{\cup}\, \{\gamma_1' \subseteq \gamma_1\}, S_2\,t, S_2\,b) \longrightarrow (S_1\,S_2\,C, S_1\,S_2\,t, S_1\,S_2\,b)$$

which is as desired since clearly $S_2\,S_1 = S_1\,S_2$. □

## Algorithm $\mathcal{W}$

**Lemma 7.2** Let $C$ be well-formed; then $C, A \vdash e : t \,\&\, b$ holds if and only if $C, A \vdash e : GEN(A, b)(C, t) \,\&\, b$.

**Proof** For "if" simply use rule (ins) with $S_0 = Id$. Next consider "only if" and write

$$\{\vec{\gamma}\} = (FV(t)^{C\updownarrow}) \backslash (FV(A, b)^{C\downarrow})$$
$$C_0 = C\mid_{\{\vec{\gamma}\}} = \{(g_1 \subseteq g_2) \in C \mid FV(g_1, g_2) \cap \{\vec{\gamma}\} \neq \emptyset\}$$

so that $GEN(A, b)(C, t) = \forall(\vec{\gamma} : C_0).\ t$; this is well-formed by Fact 3.7.

Next let $R$ be a renaming of $\{\vec{\gamma}\}$ into fresh variables. It is immediate that $\forall(\vec{\gamma} : C_0).\ t$ is solvable from $(C\backslash C_0) \cup R\,C_0$ by some $S_0$; simply take $S_0 = R$. Finally note that $\{\vec{\gamma}\} \cap FV((C\backslash C_0) \cup R\,C_0) = \emptyset$ by construction of $C_0$ and $R$, and that $\{\vec{\gamma}\} \cap FV(A, b) = \emptyset$ by construction of $\{\vec{\gamma}\}$.

We then have (using Lemma 2.3 on the assumption) that

$$((C\backslash C_0) \cup R\,C_0) \cup C_0, A \vdash e : t \ \&\ b$$

and (gen) gives

$$(C\backslash C_0) \cup R\,C_0, A \vdash e : \forall(\vec{\gamma} : C_0).\ t \ \&\ b$$

and finally Lemma 2.2 gives the desired result:

$$(C\backslash C_0) \cup C_0, A \vdash e : \forall(\vec{\gamma} : C_0).\ t \ \&\ b$$

This completes the proof. $\square$

**Theorem 7.3** If $\mathcal{W}(A, e) = (S, t, b, C)$ with $A$ simple then $C, S\,A \vdash e : t \;\&\; b$.

**Proof** We proceed by structural induction on $e$; we first prove the result for $\mathcal{W}'$ (using the notation introduced in the defining clause for $\mathcal{W}'(A, e)$) and then in a joint final case extend the result to $\mathcal{W}$.

**The case** $e ::= c$. If TypeOf($c$) is a type $t_0$ then $S = \mathrm{Id}$, $t = t_0$, $b = \emptyset$, $C = \emptyset$ and the claim is trivial. Otherwise write TypeOf($c$) $= \forall(\vec{\gamma_0} : C_0).\ t_0$, let $\vec{\gamma}$ be fresh and write $R = [\vec{\gamma_0} \mapsto \vec{\gamma}]$. Then $S = Id, t = R\,t_0, b = \emptyset$, and $C = R\,C_0$. We then have

$$\begin{array}{ll} C, A \vdash c : \forall(\vec{\gamma_0} : C_0).\ t_0 \;\&\; \emptyset & \text{by (con)} \\ C, A \vdash c : R\ t_0 \;\&\; \emptyset & \text{by (ins)} \end{array}$$

since $Dom(R) \subseteq \{\vec{\gamma_0}\}$ and $C \vdash R\,C_0$.

**The case** $e ::= x$. If $A(x)$ is a type $t_0$ then $S = \mathrm{Id}$, $t = t_0$, $b = \emptyset$, $C = \emptyset$ and the claim is trivial. Otherwise write $A(x) = \forall(\vec{\gamma_0} : C_0).\ t_0$, let $\vec{\gamma}$ be fresh and write $R = [\vec{\gamma_0} \mapsto \vec{\gamma}]$. Then $S = Id, t = R\,t_0, b = \emptyset$, and $C = R\,C_0$. We then have

$$\begin{array}{ll} C, A \vdash x : \forall(\vec{\gamma_0} : C_0).\ t_0 \;\&\; \emptyset & \text{by (id)} \\ C, A \vdash x : R\ t_0 \;\&\; \emptyset & \text{by (ins)} \end{array}$$

since $Dom(R) \subseteq \{\vec{\gamma_0}\}$ and $C \vdash R\,C_0$.

**The case** $e ::= \texttt{fn}\ x\ \texttt{=>}\ e_0$. The induction hypothesis gives

$$C_0, S_0(A[x : \alpha]) \vdash e_0 : t_0 \;\&\; b_0$$

and using $C = C_0 \,\cup\, \{b_0 \subseteq \beta\}$ and $S = S_0$ we get

$$\begin{array}{l} C, S(A[x : \alpha]) \vdash e_0 : t_0 \;\&\; b_0 \\ C, (S\,A)[x : S\,\alpha] \vdash e_0 : t_0 \;\&\; \beta \\ C, S\,A \vdash \texttt{fn}\ x\ \texttt{=>}\ e_0 : S\,\alpha \rightarrow^{\beta} t_0 \;\&\; \emptyset \end{array}$$

using first Lemma 2.3, then (sub) and finally (abs).

**The case** $e ::= e_1\ e_2$. Concerning $e_1$ the induction hypothesis gives

$$C_1, S_1\,A \vdash e_1 : t_1 \;\&\; b_1$$

Using Lemmas 2.2 and 2.3 and then (sub) we get

$$S_2\,C_1, S_2\,S_1\,A \vdash e_1 : S_2\,t_1 \;\&\; S_2\,b_1$$
$$C, S\,A \vdash e_1 : S_2\,t_1 \;\&\; S_2\,b_1$$
$$C, S\,A \vdash e_1 : t_2 \rightarrow^{\beta} \alpha \;\&\; S_2\,b_1$$

Turning to $e_2$ the induction hypothesis (which can be applied since $S_1$ and hence $S_1\,A$ is simple due to Lemma 7.1) gives

$$C_2, S_2\,S_1\,A \vdash e_2 : t_2 \;\&\; b_2$$

and using Lemma 2.3 we get

$$C, S\,A \vdash e_2 : t_2 \;\&\; b_2.$$

Finally we get

$$C, S\,A \vdash e_1\;e_2 : \alpha \;\&\; S_2\,b_1 \,\cup\, b_2 \,\cup\, \beta$$

which is the desired result.

**The case** $e ::=$ `let` $x$ `=` $e_1$ `in` $e_2$. Concerning $e_1$ the induction hypothesis gives

$$C_1, S_1\,A \vdash e_1 : t_1 \;\&\; b_1$$

and note that by Lemma 7.1 it holds that $C_1$ is well-formed. Next let $ts_1 = \mathit{GEN}(S_1\,A, b_1)(C_1, t_1)$ so that Lemmas 7.2, 2.2 and 2.3 give

$$C_1, S_1\,A \vdash e_1 : ts_1 \;\&\; b_1$$
$$S_2\,C_1, S\,A \vdash e_1 : S_2\,ts_1 \;\&\; S_2\,b_1$$
$$C, S\,A \vdash e_1 : S_2\,ts_1 \;\&\; S_2\,b_1$$

Turning to $e_2$ the induction hypothesis gives

$$C_2, (S_2\,S_1\,A)[x : S_2\,ts_1] \vdash e_2 : t_2 \;\&\; b_2$$

and using Lemma 2.3 we get

$$C, S\,A[x : S_2\,ts_1] \vdash e_2 : t_2 \;\&\; b_2$$

and hence using (let)

$$C, S\,A \vdash \texttt{let}\; x \;\texttt{=}\; e_1 \;\texttt{in}\; e_2 : t_2 \;\&\; S_2\,b_1 \,\cup\, b_2$$

and this is the desired result.

**The case** $e ::=$ `rec` $f\ x$ `=>` $e_0$. Concerning $e_0$ the induction hypothesis gives

$$C_0, S_0\,A[f : S_0\,\alpha_1 \to^{S_0\,\beta} S_0\,\alpha_2][x : S_0\,\alpha_1] \vdash e_0 : t_0 \ \&\ b_0$$

Using Lemma 2.3, (sub), (abs) and (rec) we then get

$$C, S\,A[f : S\,\alpha_1 \to^{S\,\beta} S\,\alpha_2][x : S\,\alpha_1] \vdash e_0 : t_0 \ \&\ b_0$$
$$C, S\,A[f : S\,\alpha_1 \to^{S\,\beta} S\,\alpha_2][x : S\,\alpha_1] \vdash e_0 : S\,\alpha_2 \ \&\ S\,\beta$$
$$C, S\,A[f : S\,\alpha_1 \to^{S\,\beta} S\,\alpha_2] \vdash \texttt{fn}\ x\ \texttt{=>}\ e_0 : S\,\alpha_1 \to^{S\,\beta} S\,\alpha_2 \ \&\ \emptyset$$
$$C, S\,A \vdash \texttt{rec}\ f\ x\ \texttt{=>}\ e_0 : S\,\alpha_1 \to^{S\,\beta} S\,\alpha_2 \ \&\ \emptyset$$

which is the desired result.

**The case** $e ::=$ `if` $e_0$ `then` $e_1$ `else` $e_2$. The induction hypothesis, Lemmas 2.2 and 2.3 and rule (sub) give:

$$C, S\,A \vdash e_0 : \texttt{bool} \ \&\ S_2\,S_1\,b_0$$
$$C, S\,A \vdash e_1 : \alpha \ \&\ S_2\,b_1$$
$$C, S\,A \vdash e_2 : \alpha \ \&\ b_2$$

and rule (if) then gives

$$C, S\,A \vdash \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \alpha \ \&\ S_2\,S_1 b_0 \cup S_2\,b_1 \cup b_2$$

which is the desired result.

**Lifting the result from $\mathcal{W}'$ to $\mathcal{W}$.** We have from the above that $\mathcal{W}'(A, e) = (S_1, t_1, b_1, C_1)$ with $C_1$ simple and that

$$C_1, S_1\,A \vdash e : t_1 \ \&\ b_1$$

Concerning $\mathcal{F}$ we have

$$(S_2, C_2) = \mathcal{F}(C_1)$$

where Lemma 4.6 and Lemma 4.7 ensure that $C_2$ is simple, well-formed and atomic and that $C_2 \vdash S_2\ C_1$. Using Lemmas 2.2 and 2.3 we get

$$C_2, S_2\,S_1\,A \vdash e : S_2\,t_1 \ \&\ S_2\,b_1$$

Concerning $\mathcal{R}$ we have

$$(C_3, t_3, b_3) = \mathcal{R}(C_2, S_2\, t_1, S_2\, b_1, S_2\, S_1\, A)$$

so by Lemma 5.13 we get

$$C_3, S_2\, S_1\, A \vdash e : t_3 \ \&\ b_3$$

which is the desired result. $\square$