Polymorphic Subtyping for Effect Analysis: the Algorithm

Flemming Nielson & Hanne Riis Nielson & Torben Amtoft

Computer Science Department, Aarhus University, Denmark

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

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 we define a type inference algorithm and then construct procedures for constraint normalisation and simplification. Next these algorithms are proved syntactically sound with respect to the annotated type and effect system.

1 Introduction

In a recent paper [11] we developed an annotated type and effect system for a fragment of Concurrent ML [12]. Judgements are of form

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

with e an expression, b a "behaviour", σ a type or a type scheme (to incorporate ML-style `let`-polymorphism), C a set of constraints among types and behaviours, and A an environment. Behaviours record channel allocations and thereby the set of "dangerous variables", cf. [19] (where these are called "imperative" variables) and [21]. A subtyping relation is defined using a subeffecting relation on behaviours, with the usual contravariant ordering for function space but there is no inclusion between base types.

In our approach types are annotated with behaviour information (for example the function type $t_1 \rightarrow^b t_2$). Compared with the literature we do not attempt to collect information about the *regions* in which communications take place [17, 20]; neither do we enable polymorphic recursion in the type annotations as in [4, 20]; on the other hand we *do* allow behaviours to contain annotated types.

In [11] it was demonstrated that one must be very careful when deciding the set of variables over which to generalise in the inference rule for `let`: not only should this set be disjoint from the set of variables occurring in the behaviour (as is standard in effect systems, e.g. [17]) but it should also be *upwards closed* with respect to a constraint set. The semantic soundness of the generalisation rule (and of the overall system) was proved in [1] and relied heavily on the upwards closure.

The goal of this paper is to produce a type reconstruction algorithm in the spirit of Milner's algorithm $\mathcal{W}$ [7]: given an expression e and an environment A, the recursively defined function $\mathcal{W}$ will produce a substitution S, a type t,

and a behaviour b. The definition in [7] employs unification: if e_1 has been given type $t_0 \to t_1$ and e_2 has been given type t_2 then in order to type $e_1\, e_2$ one must unify t_0 and t_2. Unification works by decomposition: in order to unify $t_1 \to t_2$ and $t_1' \to t_2'$ one recursively unifies t_1 with t_1' and t_2 with t_2'. Decomposition is valid because types constitute a "free algebra": two types are equal if and only if they have the same top-level constructor and also their subcomponents are equal. However, this will not be the case for behaviours, and therefore $\mathcal{W}$ of [7] cannot immediately be generalised to work on annotated types.

We thus have to rethink the unification algorithm. As the behaviours of this paper satisfy certain associativity and commutativity properties one might adapt results from unification theory [13] to get a unification algorithm producing a set of unifiers from which all other unifiers can be derived; but as we shall aim at a theory which facilitates extension to more complex behaviours (such as those presented in [10]) where it seems unlikely that we can use results from unification theory, we shall refrain from such attempts and instead follow [6] and generate *behaviour constraints*: that is, in the process of unifing $t_1 \to^b t_2$ and $t_1' \to^{b'} t_2'$ we generate constraints relating b and b'.

In order to incorporate subtyping we also need to generate *type constraints* as in [3, 15] (in these papers there may also be inclusions between base types such as $\texttt{int} \subseteq \texttt{real}$). The presence of type constraints as well as behaviour constraints is a consequence of our overall design: types and behaviours should be inferred simultaneously "from scratch", as is done by the algorithm $\mathcal{W}$ presented in Sect. 3. This should be compared with the approach in [18] where an effect system with subtyping but without polymorphism is presented; as the "underlying" types are given in advance it is sufficient to generate behaviour constraints.

The constraints generated by $\mathcal{W}$ have to be massaged so as to satisfy certain invariants ("atomicity" and "simplicity") and for this we devise the algorithm $\mathcal{F}$ (Sect. 4), inspired by [3]. Still the algorithm will produce a rather unwieldy number of constraints; to reduce this number dramatically we may apply an algorithm $\mathcal{R}$ (defined in Sect. 5) which adapts the techniques of [2, 15]. The resulting algorithm $\mathcal{W}$ (which uses $\mathcal{F}$ and $\mathcal{R}$) has been implemented, and in Sect. 6 we shall see that the output produced by our prototype is quite readable and informative.

Soundness. In Sect. 7 we shall prove that $\mathcal{W}$ is (syntactically) sound, that is if $\mathcal{W}(A, e) = (S, t, b, C)$ then $C, S\,A \vdash e : t \,\&\, b$; the completeness of $\mathcal{W}$ is still an open question and we do not attempt to estimate the complexity of the algorithm.

As the main distinguishing feature of our inference system (as mentioned above), essential for semantic soundness, was the choice of generalisation rule; so the distinguishing feature of our algorithm, essential for syntactic soundness (and which still leaves hope for completeness), is the choice of generalisation rule. This involves (much as in [21]) taking downwards closure of a set of variables with respect to a constraint set.

In Sect. 8 we shall see that the constraints generated by $\mathcal{W}$ can always be solved provided recursive behaviour systems are admitted. Alternatively recur-

sive 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 λ-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 [11]. Expressions and constants are given by

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

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` for creating delayed output and input communications, and the non-sequential base functions `sync` (for synchronising a delayed communication), `channel` (for allocating a new communication channel) and `fork` (for spawning a new process).

Types and behaviours are given by

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

where a type $t_1 \rightarrow^b t_2$ denotes a function which given a value of type t_1 computes a value of type t_2 with "side effect" b; where a type t `chan` denotes a channel allowing values of type t to be transmitted; and where a type t `com` b denotes a delayed communication that when synchronised will give rise to side effect b and result in a type t. A behaviour can (apart from the presence of behaviour variables) be viewed as a set of "atomic" behaviours which each record the creation of a channel.

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.

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 $\{t\ \textsc{chan}\} \subseteq \{t'\ \textsc{chan}\}$ 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 defined in Figure 3. The rules for abstraction and application are as usual in effect systems: the latent behaviour of the body of

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.

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 recursion makes use of function abstraction to concisely represent the "fixed point requirement" of typing recursive functions; note that we do not admit polymorphic recursion. The rule (sub) makes use of the subtyping and subeffecting relation defined in Fig. 2. The rule (ins) allows one to instantiate a type scheme and employs the notion of solvability: 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 $C \vdash S_0\, C_0$, where we write $C \vdash C'$ to mean that[1] $C \vdash g_1 \subseteq g_2$ for all $g_1 \subseteq g_2$ in C'.

The generalisation rule (gen) employs the notion of *well-formedness* where we need several auxiliary concepts: the judgement $C \vdash \gamma_1 \leftarrow \gamma_2$ holds if there exists $(g_1 \subseteq g_2)$ in C such that $\gamma_i \in FV(g_i)$ for $i = 1, 2$; the relation $\to$ is the inverse of $\leftarrow$; the relation $\leftrightarrow$ is the *union* of $\leftarrow$ and $\to$; and as usual $\leftarrow^*$ (respectively $\to^*$ and $\leftrightarrow^*$) denotes the reflexive and transitive closure of these relations. We then define *upwards closure*, *downwards closure*, and *up-downwards closure*:

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

and are then ready to define well-formedness for constraints and for type schemes:

[1] We use g to range over t or b as appropriate and γ to range over α and β as appropriate and σ to range over t and ts as appropriate.

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. 2. Subtyping and subeffecting.

Definition 1. A constraint set is well-formed if all constraints are of form $t \subseteq \alpha$ or $b \subseteq \beta$.

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 it is *upwards closed*: that is $\{\vec{\alpha}\vec{\beta}\}^{C_0\uparrow} = \{\vec{\alpha}\vec{\beta}\}$.

A type t is trivially well-formed. □

Requiring a type scheme to be well-formed (in particular upwards closed) is a crucial feature in the approach of [11], the semantic soundness of which was established in [1].

(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. 3. The type inference system.

As expected (and proved in [11]) we have the following results, crucial for showing soundness of our inference algorithm, saying that we can apply a substitution or strengthen the constraint set, and still get a valid judgement:

Lemma 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 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 (cf. the Introduction) motivated by the overall approach of [15, 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. The third ingredient (called $\mathcal{R}$) amounts to reducing the constraint set; this is optional and a somewhat open ended endeavour.

3.1 Well-formedness, Simplicity and Atomicity

In Definition 1 we introduced the notion of well-formedness for constraint sets and type schemes; for use by the algorithm we shall in addition introduce the notions of *simplicity* and *atomicity*.

Simplicity. As we do not know how to solve general behaviour constraints (for example we have no techniques available for decomposing constraints of form $\beta_1 \subseteq \beta_2 \cup \beta_3$) we shall wish to maintain the invariant that all (behaviour) constraints which arise in the algorithm must be well-formed. In order to achieve this we must follow [16] and [21] and demand that all types in question are (what [9] calls) *simple*, that is all behaviour annotations are variables (as is the case for $\texttt{int} \rightarrow^{\beta} \texttt{int}$).

Definition 4. A type is simple if all its behaviour annotations are behaviour variables; a constraint set is simple if all type constraints are of form $(t_1 \subseteq t_2)$ with t_1 and t_2 simple and if all behaviour constraints are of form $(b \subseteq \beta)$; a type scheme is simple if the constraint set and the type both are; an environment is simple if all its type schemes are; finally a substitution is simple if it maps behaviour variables to behaviour *variables* and type variables to simple types.

That is, with st ranging over simple types we have

$$st ::= \cdots \mid st_1 \rightarrow^{\beta} \ st_2 \mid st_1 \times st_2 \mid \cdots st \ \mathtt{com} \ \beta$$

Fact 5. Let t be a simple type, 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, $S\,C$ is a simple constraint set, $S\,ts$ is a simple type scheme, and $S'\,S$ is a simple substitution.

There is a discrepancy between the inference system and the algorithm in the sense that non-simple types are allowed in the former, in order to increase expressibility, but disallowed in the latter. In [21] a "direct" as well as an "indirect" inference system is presented; the "indirect" is geared towards an algorithm and requires simplicity and employs constraints. A perhaps more uniform approach is given in [20] where arrows are annotated with pairs of form $\epsilon.\phi$ with ϵ an effect variable and with ϕ a set of region or effect variables; one can think of this as an arrow annotated by ϵ *together with* the constraint $\phi \subseteq \epsilon$.

Atomicity. As in [8, 3, 15] we shall want the type constraints to *match* and shall decompose them into *atomic* constraints; in our setting these will not contain base types as we have no ordering among those.

Definition 6. A constraint set is atomic if all type constraints are of form $\alpha_1 \subseteq \alpha_2$ and if all behaviour constraints are of form $\{t \ \textsc{chan}\} \subseteq \beta$ or $\beta' \subseteq \beta$.

Fact 7. An atomic constraint set is also well-formed and simple.

Requiring a well-formed set of behaviour constraints to be atomic is unproblematic because a constraint $(\emptyset \subseteq \beta)$ can always be thrown away and a constraint $(b_1 \cup b_2 \subseteq \beta)$ can always be split to $(b_1 \subseteq \beta)$ and $(b_2 \subseteq \beta)$. Requiring atomicity of type constraints is responsible for transforming constraints like $(\mathtt{int} \subseteq \alpha)$ and $(t_1 \times t_2 \subseteq \alpha)$ by forcing α to be replaced by a type expression that "matches" the left hand side, and for disallowing constraints like $(t_1 \times t_2 \subseteq t_1' \rightarrow^{b} \ t_2')$. This feature is responsible for making the algorithm a "conservative extension" 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. (We shall make this point more precise at the end of Section 7.)

3.2 Algorithm $\mathcal{W}$

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

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

and we shall aim at defining it such that $C, S\,A \vdash e : t \,\&\, b$ holds (soundness) and such that we might hope for some notion of principality (completeness); here completeness might be taken to mean that if also $C', S''\,A \vdash e : t' \,\&\, b'$ then

there exists S' such that $C' \vdash S'\,C$ and $C' \vdash S'\,t \subseteq t'$ and $C' \vdash S'\,b \subseteq b'$ and $S'\,S$ equals S'' on A. It is an open question whether such a principal typing exists and, if this is the case, whether our algorithm actually computes this principal typing.

We shall enforce throughout that if A is simple then all of S, t and C are simple and that C is atomic. Algorithm $\mathcal{W}$ is defined by the clause

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

where the definitions of $\mathcal{W}'$ and $\mathcal{W}$ are mutually recursive; algorithm $\mathcal{W}'$ is responsible for the syntax-directed traversal of the argument expression e. In general, $\mathcal{W}'$ will fail to produce an atomic constraint set C, even when the environment A is well-formed and simple; it will be the case, however, that all of S_1, t_1, and C_1 are simple. This then motivates the need for a transformation $\mathcal{F}$ (Section 4) that maps a simple constraint set into an atomic (hence also well-formed and simple) constraint set; since this involves splitting variables we shall need to produce a simple 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 1. To make the intentions a bit clearer suppose $\mathcal{W}'(A,e) = (S_1,t_1,b_1,C_1)$ so that $C_1, S_1\,A \vdash e : t_1 \,\&\, b_1$ is a typing of e. If

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

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

$$C_2 = \{\alpha_1 \subseteq \alpha_{31},\ \alpha_2 \subseteq \alpha_{32},\ \{\alpha_5\ \textsc{chan}\} \subseteq \beta\}$$

$$S_2 = [\alpha_3 \mapsto \alpha_{31} \times \alpha_{32},\, \alpha_4 \mapsto \texttt{int}]$$

Here we expand α_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 atomic. Furthermore we have expanded α_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 β into two and then removed the trivial $\emptyset \subseteq \beta$ constraint. The intention is that also $C_2, S_2\,S_1\,A \vdash e : S_2\,t_1 \,\&\, S_2\,b_1$ is a typing of e; additionally the constraint set is atomic (unlike what is the case for C_1); and we have not lost any principality when going from the former typing to the latter. □

3.3 Algorithm $\mathcal{W}'$

Algorithm $\mathcal{W}'$ is defined by the clauses in Figure 4 and 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 [15, 5] 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

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

$$\mathit{INST}(t) = (\mathrm{Id}, t, \emptyset, \emptyset)$$

in order to produce a fresh instance of the relevant type or type scheme (as determined from TypeOf or from A); in order to ensure that this instance is simple (as will be needed if it is determined from TypeOf) we also need the funktion *Mk-simple* which replaces all occurrences of $\emptyset$ in a type by fresh behaviour variables, for instance we have $\mathit{Mk\text{-}simple}(\texttt{int} \to^{\emptyset} \texttt{int} \to^{\emptyset} \texttt{int}) = \texttt{int} \to^{\beta_1} \texttt{int} \to^{\beta_2} \texttt{int}$ with β_1 and β_2 fresh. As all behaviour annotations in Fig. 1 are either $\emptyset$ or variables, and as the former kind of annotation occurs only in top-level positive position (that is, not on the left hand side of a function arrow or inside some channel type), the following property holds:

Fact 8. Let c be a constant, and let $\mathit{INST}(\mathrm{TypeOf}(c)) = (S, t, b, C)$. Then C is a simple constraint set, $\mathit{Mk\text{-}simple}(t)$ is a simple type and $\emptyset \vdash t \subseteq \mathit{Mk\text{-}simple}(t)$.

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:

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

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.

[2] Interestingly, this is exactly the place where the algorithm of [15] 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.

$$\mathcal{W}'(A, c) = \text{if } c \in \mathit{Dom}(\text{TypeOf})$$
$$\quad \text{then let } (S, t, b, C) = \mathit{INST}(\text{TypeOf}(c))$$
$$\quad\quad \text{in } (S, \mathit{Mk\text{-}simple}(t), b, C)$$
$$\quad \text{else } \mathit{fail}_{const}$$

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

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

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

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

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

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

Fig. 4. Syntax-directed constraint generation.

The exclusion of the set $FV(A,b)^{C\downarrow}$ (rather than just $FV(A,b)$) is necessary in order to ensure $\{\vec{\alpha}\vec{\beta}\}^{C_0\uparrow} = \{\vec{\alpha}\vec{\beta}\}$ which (cf. the remarks concerning Def. 1) is essential for semantic soundness; the computation of "Indirect Free Variables" of [21] is very similar to our notion of downwards closure. 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 may still hope for syntactic completeness and all of $FV(t)$, $FV(t)^{C\downarrow}$ (similar to what is in fact taken in [21]) and $FV(t)^{C\uparrow}$ are apparently too small for this (except for the latter they are not even upwards closed).

Fact 9. Let $\sigma = GEN(A,b)(C,t)$; if C is well-formed then so is σ.

Proof. The only non-trivial task is to show that $\{\vec{\alpha}\vec{\beta}\}^{C_0\uparrow} \subseteq \{\vec{\alpha}\vec{\beta}\}$ where $\{\vec{\alpha}\vec{\beta}\}$ is as in the defining clause for *GEN*. So assume $C_0 \vdash \gamma_1 \leftarrow \gamma_2$ (and hence also $C \vdash \gamma_1 \leftarrow \gamma_2$) 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. □

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 sensible only 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. □

4 Algorithm $\mathcal{F}$

We are now going to define the algorithm $\mathcal{F}$ which "forces type constraints to match" by transforming them into atomic constraints; the algorithm closely resembles [3, procedure MATCH].

The algorithm 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 Eq_C for the identity relation over type variables in C. We then define $\mathcal{F}$ by

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

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

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

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

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

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

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

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

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

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

Fig. 5. Decomposition of constraints.

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

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

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

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

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 10. 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 and produces a new substitution as a result; additionally it may modify the equivalence relation among type variables. The details require the predicate $\mathcal{M}$ (which performs an "occur check"), to be defined shortly. Before presenting the formal definition we consider an example.

Example 2. Consider the constraint $t_1 \subseteq \alpha_0$ where $t_1 = (\alpha_{11} \times \alpha_{12})\ \mathtt{com}\ \beta_1$. 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 α_{21} and α_{22} and a new behaviour variable β_2 and by defining

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

Then $R\,t_1 = t_1 = (\alpha_{11} \times \alpha_{12})\ \mathtt{com}\ \beta_1$ and $R\,\alpha_0 = (\alpha_{21} \times \alpha_{22})\ \mathtt{com}\ \beta_2$ 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 $(\alpha \times \alpha)\ \mathtt{com}\ \beta \subseteq \alpha$ then the above procedure would not lead to a matching constraint. We would get

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

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

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

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 "occur check": when $\mathcal{M}(\alpha, t, \sim, R, \sim')$ holds no variable in $Dom(R)$ must occur in t. This condition fails when $t = (\alpha \times \alpha)\ \mathtt{com}\ \beta$.

However, 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 "occur 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 follow [3] in making use of the equivalence relation $\sim$ and extend it with $\alpha_1 \sim \alpha_2$ after the first match that produced R_1; the intuition is that α_1 and α_2 eventually must be bound to types having the same shape. When performing the second match we then require R_2 not only to expand α_1 but also all α' satisfying $\alpha' \sim \alpha_1$; this means that R_2 must expand also α_2. Consequently the "extended occur check" $Dom(R_2) \cap FV(\alpha_2 \texttt{ com } \beta_2) = \emptyset$ fails. □

Remark. Matching bears certain similarities to unification and can actually be defined in terms of unification. In [8] matching is performed by first doing unification and then the resulting substitution is transformed such that it "maps into fresh variables". In [14, Fig. 3.7] it is first checked whether it is possible to unify a certain set of equations, derived from the constraint set; if this is the case then the algorithm behaves similar to the one presented here except that the equivalence relation is no longer needed. □

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 11. A shape sh is a type with holes in it for all type variables and for all behaviours; it may be formally defined by:

$$\begin{aligned} sh ::= {} & [\,] \mid \texttt{ unit} \mid \texttt{int} \mid \texttt{bool} \mid sh_1 \times sh_2 \mid sh_1 \rightarrow^{[]} sh_2 \mid sh \texttt{ list} \\ & \mid sh \texttt{ chan} \mid sh \texttt{ com } [\,] \end{aligned}$$

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 replacing all behaviour holes with the relevant behaviour in the list $\vec{b}$; we assume throughout that the lengths of the lists equal the number of holes and shall dispense with a formal definition. For each type t there clearly exists unique sh, $\vec{t}$ and $\vec{b}$ such that $t = sh[\vec{t}, \vec{b}]$, and t is simple if and only if all elements of $\vec{b}$ are variables.

Example 3. If $sh = ([\,] \times [\,])$ com $[\,]$ then $sh[\vec{t}, \vec{b}] = (t_1 \times t_2)$ com b_1 if and only if $\vec{t} = t_1 t_2$ and $\vec{b} = b_1$. □

The axioms (mr) and (ml) force a type t to match a type variable α and employ the predicate $\mathcal{M}$ defined in Figure 7. This predicate may also be considered a partial function with its first three parameters being input and the last two being output; the "call" $\mathcal{M}(\alpha, t, \sim, R, \sim')$ produces the substitution R and modifies the equivalence relation $\sim$ (over the free type variables of a constraint set C') to another equivalence relation $\sim'$ (over the free type variables of the constraint set $R\,C'$). In axioms (mr) and (ml) the newly produced substitution R is composed with the previously produced substitution. Also note that the "extended occur check" in Figure 7 ensures that $Rt = t$. Using Fact 5 it is straightforward to establish:

$\mathcal{M}(\alpha, t, \sim, R, \sim')$ holds
if $\{\alpha_1, \cdots, \alpha_n\} \cap FV(t) = \emptyset$
and $R = [\alpha_1 \mapsto sh[\vec{\alpha}_1, \vec{\beta}_1], \ldots, \alpha_n \mapsto sh[\vec{\alpha}_n, \vec{\beta}_n]]$
and $\sim'$ is the least equivalence relation containing the pairs
$\{(\alpha', \alpha'') \mid \alpha' \sim \alpha'' \wedge \{\alpha', \alpha''\} \cap \{\alpha_1, \cdots, \alpha_n\} = \emptyset\} \bigcup$
$\{(\alpha_{0j}, \alpha_{ij}) \mid \vec{\alpha}_0 = \alpha_{01} \cdots \alpha_{0m}, \vec{\alpha}_i = \alpha_{i1} \cdots \alpha_{im}, 1 \leq i \leq n, 1 \leq j \leq m\}$
where $\{\alpha_1, \cdots, \alpha_n\} = \{\alpha' \mid \alpha' \sim \alpha\}$
and $sh[\vec{\alpha}_0, \vec{\beta}_0] = t$ with $\vec{\alpha}_0$ having length m
and $\vec{\alpha}_1, \cdots, \vec{\alpha}_n$ are vectors of fresh variables of length m
and $\vec{\beta}_1, \cdots, \vec{\beta}_n$ are vectors of fresh variables of the same length as $\vec{\beta}_0$

Fig. 7. Forced matching for simple types.

Fact 12. Suppose $(S, C, \sim) \longrightarrow (S', C', \sim')$. Then there exists simple R such that $S' = R\,S$ and such that $R\,C \rightarrowtail^* C'$. 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

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

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

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

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

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

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

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

which cannot be rewritten further. The set C' is atomic so Algorithm $\mathcal{F}$ succeeds on C. □

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

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

Lemma 13. $\mathcal{F}(C)$ always terminates (possibly with failure). If $\mathcal{F}(C) = (S', C')$ then

- if C is simple then S' is simple and C' is atomic;
- C' is determined from $S'\,C$ in the sense that $S'\,C \rightharpoonup^* C' \not\rightharpoonup$.

Proof. We first address termination and for this purpose we (much as in [3]) define an ordering on triples $(S, C, \sim)$ as follows: $(S', C', \sim')$ is less than $(S, C, \sim)$ if *either* the number of equivalence classes in $FV(C')$ wrt. $\sim'$ is less than the number of equivalence classes in $FV(C)$ wrt. $\sim$ *or* these numbers are equal but C' is less than C according to the following definition:

> for all $i \geq 0$ let s_i be the number of constraints in C containing i symbols and let s'_i be the number of constraints in C' containing i symbols; then C' is less than C if there exists a n such that $s'_n < s_n$ and such that $s'_i = s_i$ for all $i > n$.

This relation on constraint sets is clearly transitive and it is easy to see that it is also well-founded, hence the (lexicographically defined) ordering on triples is well-founded. Thus it suffices to show that if $(S, C, \sim) \longrightarrow (S', C', \sim')$ then $(S', C', \sim')$ is less than $(S, C, \sim)$. If the rule (dc) has been applied then C' is less than C (as n in the above definition we can use the number of symbols in the constraint being decomposed) and $\sim' = \sim$. If the rule (mr) or (ml) has been applied then the number of equivalence classes wrt. $\sim$ will decrease as can be seen from the definition of $\mathcal{M}$ in Fig. 7: the equivalence class containing α is removed (as this class equals $Dom(R)$ and $C' = R\,C$) and no new classes are added (as all type variables in $Ran(R)$ are put into some existing equivalence class).

We have thus proved termination; 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 12) 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 12 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 14. *$\mathcal{F}$ is sound*

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

Proof. By Lemma 13 we have $S'\,C \rightharpoonup^* C'$, which yields the claim due to Fact 10.

Remark. By Fact 10 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 13 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 atomic. The techniques used are basically those of [15] 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:

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

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 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 15. We write $C \vdash \gamma \Leftarrow^* \gamma'$ if there is a path from γ' to γ: that is 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 $C \vdash \gamma \Leftarrow^* \gamma$ holds also if $\gamma \notin FV(C)$. From reflexivity and transitivity of $\subseteq$ we have:

Fact 16. If $C \vdash \gamma \Leftarrow^* \gamma'$ then also $C \vdash \gamma \subseteq \gamma'$.

We have a substitution result similar to Lemma 2:

Fact 17. Let S be a substitution mapping variables into variables, and suppose $C \vdash \gamma \Leftarrow^* \gamma'$. Then also $S\,C \vdash S\,\gamma \Leftarrow^* S\,\gamma'$.

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

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 [15] and is essential for a compact representation of the constraints.

The remaining rules all replace some variable γ by another variable γ'. However, it is not the case that all substitutions that solve C can be written on the form $S'\,[\gamma \mapsto \gamma']$ and therefore we might lose completeness if the substitution

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

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

(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 γ

Fig. 8. Eliminating constraints.

$[\gamma \mapsto \gamma']$ was to be returned (in the style of $\mathcal{F}$) and subsequently applied to A. In order to maintain soundness we must therefore demand that $\gamma \notin FV(A)$.

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. (In [14] it holds that $\emptyset \vdash t_1 \equiv t_2$ implies $t_1 = t_2$ so if γ and γ' belong to the same cycle in C then all substitutions that solve C can be written on the form $S'\,[\gamma \mapsto \gamma']$, hence cycle elimination can be part of the analogue of $\mathcal{F}$.)

The rule (shrink) expresses that a variable γ can be replaced by its "immediate predecessor" γ', and due to the ability to perform transitive reduction this can be strengthened to the requirement that γ' is the "only predecessor" of γ, which can be formalised as the side condition $\gamma \notin FV(RHS(C))$ where $RHS(C) = \{\gamma \mid \exists g : (g \subseteq \gamma) \in C\}$. We can allow γ to belong to t and b and $LHS(C)$, where $LHS(C) = \{g \mid \exists \gamma : (g \subseteq \gamma) \in C\}$, 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 $g \in LHS(C)$. This will be the case provided t and b and each element of $LHS(C)$ are *monotonic* in γ, where for example $t = \alpha_1 \rightarrow^{\beta_1} \alpha_2 \rightarrow^{\beta_1} \alpha_1$ (recall that $\rightarrow$ is right-associative) is monotonic in γ 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 γ.

The rule (boost) expresses that a variable γ can be replaced by its "immediate successor" γ', and due to the ability to perform transitive reduction this can be strengthened to the requirement that γ' must be the "only successor" of γ. 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[3] b and each element of $\mathit{LHS}(C)$ are *anti-monotonic* in γ, where for example $t = \alpha_1 \to^{\beta_1} \alpha_2 \to^{\beta_1} \alpha_1$ is anti-monotonic in γ for *all* $\gamma \notin \{\alpha_1, \beta_1\}$. Notice that if each element of $\mathit{LHS}(C)$ is anti-monotonic in γ then γ' in fact is the only successor of γ.

Example 4. 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 α_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

$$\{t_1 \to^{b} t_2 \mid \emptyset \vdash t_1 \subseteq t_2\} \text{ and } b \text{ is a behaviour}$$

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 α_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. 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 α_1 nor anti-monotonic in α_2.

Monotonicity

Definition 18. Given a constraint set C. We say that a substitution S is increasing (respectively decreasing) wrt. C if for all γ 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:

[3] Actually, the condition that b is anti-monotonic in γ amounts to $\gamma \notin \mathit{FV}(b)$. Similarly with the requirement on $\mathit{LHS}(C)$ (as C is atomic).

Lemma 19. Suppose that g is monotonic in all $\gamma \in Dom(S)$; 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 $\gamma \in Dom(S)$; 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 $NA(g)$ and $NM(g)$ (for "Not Anti-monotonic" and "Not Monotonic"):

$NA(\gamma) = \{\gamma\}$ and $NM(\gamma) = \emptyset$;
$NA(\texttt{unit}) = NA(\texttt{int}) = NA(\texttt{bool}) = NA(\emptyset) = \emptyset$;
$NM(\texttt{unit}) = NM(\texttt{int}) = NM(\texttt{bool}) = NM(\emptyset) = \emptyset$;
$NA(t_1 \to^b t_2) = NM(t_1) \cup NA(b) \cup NA(t_2)$;
$NM(t_1 \to^b t_2) = NA(t_1) \cup NM(b) \cup NM(t_2)$;
$NA(t_1 \times t_2) = NA(t_1) \cup NA(t_2)$ and $NM(t_1 \times t_2) = NM(t_1) \cup NM(t_2)$;
$NA(t\ \texttt{list}) = NA(t)$ and $NM(t\ \texttt{list}) = NM(t)$;
$NA(t\ \texttt{chan}) = NM(t\ \texttt{chan}) = FV(t)$;
$NA(t\ \texttt{com}\ b) = NA(t) \cup NA(b)$ and $NM(t\ \texttt{com}\ b) = NM(t) \cup NM(b)$;
$NA(\{t\ \text{CHAN}\}) = NM(\{t\ \text{CHAN}\}) = FV(t)$;
$NA(b_1 \cup b_2) = NA(b_1) \cup NA(b_2)$ and $NM(b_1 \cup b_2) = NM(b_1) \cup NM(b_2)$.

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

Definition 20. We say that g is monotonic in γ if $\gamma \notin NM(g)$; and we say that g is anti-monotonic in γ if $\gamma \notin NA(g)$.

Fact 21. For all types and behaviours g, it holds that $NM(g) \cup NA(g) = FV(g)$. (So if g is monotonic as well as anti-monotonic in γ, then $\gamma \notin FV(g)$.)

It does not necessarily hold that $NM(g)$ and $NA(g)$ are disjoint (we have e.g. $\alpha \in NM(\alpha\ \texttt{chan}) \cap NA(\alpha\ \texttt{chan})$). Now we can prove Lemma 19:

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 $\gamma \in Dom(S)$, then $g \notin Dom(S)$.

g is a function type $t_1 \to^b t_2$: First consider the sub-case where g is monotonic in all $\gamma \in Dom(S)$ and where S is increasing wrt. C. Then $\gamma \in Dom(S)$ gives $\gamma \notin NM(t_1 \to^b t_2)$, and we infer that $\gamma \notin NA(t_1)$, $\gamma \notin NM(b)$, and $\gamma \notin NM(t_2)$, so that t_1 is anti-monotonic in γ whereas t_2 and b are monotonic in γ. 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. □

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

Lemma 22. The algorithm $\mathcal{R}$ always terminates successfully. If $\mathcal{R}(C, t, b, A) = (C', t', b')$ where C is atomic and t is simple then also C' is atomic and t' is simple.

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 atomicity and simplicity. □

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

Lemma 23. 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 16. 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 and Lemma 3 we then get:

Corollary 24. 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 25. 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

No new paths are introduced in the graph by applying $\longrightarrow$:

Lemma 26. Suppose $A \vdash (C', t', b') \longrightarrow (C'', t'', b'')$ and let $\gamma_1, \gamma_2 \in FV(C'')$. Then $C' \vdash \gamma_1 \Leftarrow^* \gamma_2$ holds iff $C'' \vdash \gamma_1 \Leftarrow^* \gamma_2$ holds.

Proof. See Appendix A.

Observation 27. 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 γ_0, γ_1 and γ_2 with $\gamma_0, \gamma_1 \in FV(A)$. Then we can apply (cycle) to map γ_2 into either γ_0 or γ_1 but then we are stuck and the graphs will be different (due to the arrows to or from γ_2) unless we devise some notion of graph equivalence.

On the other hand, we have the following result:

Proposition 28. 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 atomic. Then there exists (C_1', t_1', b_1') and (C_2', t_2', b_2'), which are equal up to renaming, such that

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

(here $\longrightarrow^{\leq 1}$ denotes that either $\longrightarrow$ or $=$ holds).

Proof. See Appendix A.

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 β by its "immediate predecessor" b' even if b' is not a variable:

(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 β and
provided that $S\,t$ is 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 β and
provided that $S\,t$ is simple

Fig. 9. Additional simplifications.

for (shrink-chan) b' 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 simplicity of the type and we have explicit clauses for ensuring this; we also must preserve atomicity of the constraint set 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 22 and Lemma 23 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 would no longer hold in general that the substitution S used in Lemma 23 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 [11] 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 some 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 is 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 β_{33}, α_{49}, and α_{59} and to boost α_{58} such that we end up with one constraint only:

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

where $\alpha_{44} \subseteq \alpha_{45}$

This is small enough to be manageable and is actually more precise than the type

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

(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 29. $\mathcal{W}(A, e)$ and $\mathcal{W}'(A, e)$ always terminate (possibly with failure). If A is simple then the following holds:

- if $\mathcal{W}(A, e) = (S, t, b, C)$ then t and S are simple and C is atomic;
- if $\mathcal{W}'(A, e) = (S, t, b, C)$ then t and S as well as C are simple;
- all subcalls to $\mathcal{W}$ and $\mathcal{W}'$ are made with a simple environment.

Proof. This result is proved by structural induction in e where for constants we employ Fact 8; to lift the results from $\mathcal{W}'$ to $\mathcal{W}$ we employ Lemma 13 and Lemma 22; and throughout we employ Fact 5. □

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

Lemma 30. 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 31. 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.

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 may hope (as a weak completeness property) that then also ML typing would have failed.

On the other hand, suppose that $\mathcal{W}$ succeeds on some program e; that is we have (with $[\,]$ the empty environment)

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

which by Theorem 31 and Lemma 29 implies that

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

We then have:

– **Semantic Soundness.** It is possible[4] to apply the subject reduction result [1, Theorem 41] which (loosely speaking) says:

  all configurations which arise during execution of e can be typed.

– **Conservative extension.** Let S' be a substitution which unifies all type variables and let C^b be the behaviour constraints of C; from Lemma 2 and Lemma 3 we get $S'\,C^b, [\,] \vdash e : S'\,t \,\&\, S'\,b$ so [11, Theorem 21] (loosely speaking) tells us that if e is a sequential ML-program then

  e can be assigned a type using the ML type system.

8 Solvability of the Constraints Generated

Typability of an expression e might be taken to mean

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

and to check for typability it is natural to perform a call $\mathcal{W}([\,], e)$ and to determine whether or not the call

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

rather than with failure (recalling that by Lemma 29 the call $\mathcal{W}([\,], e)$ must terminate). As completeness issues are outside the scope of this paper we shall only ask 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 31 together with Lemma 29 that $C, [\,] \vdash e : t \,\&\, b$ with C atomic; but to achieve typability we must achieve an empty constraint set. Due to the substitution and entailment lemmas (2 and 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:

$$\text{Given atomic } C\text{; find } S' \text{ such that } \emptyset \vdash S'\,C. \tag{2}$$

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 atomicity of C this takes care of all constraints of C (provided we demand that S'_1 and S'_2 preserve atomicity).

A crude approach to defining S'_1 is to select a unique type variable α_* and let S'_1 map all type variables of $FV(C)$ to α_*. 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

[4] As C is well-formed and $[\,]$ is a "channel environment", and as we can assume by [11, Lemma 16] that the inference is "normalised".

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 α_1 with α_2.)

A crude approach to defining S_2' is to select a unique behaviour variable β_* and to let S_2' map all behaviour variables of $FV(S_1'\,C)$ to β_*. 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 [17] but this is best combined with the construction of S_3' below.)

The remaining non-trivial constraints in $S_2'\,S_1'\,C$ are of form $\{t_i\ \text{CHAN}\} \subseteq \beta_*$ with $i \in \{1 \ldots n\}$ and $n \geq 0$. If β_* does not occur in any of $t_1, \cdots, t_n$ we could follow [17] and define S_3' by letting it map β_* 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 [17] where the type inference algorithm enforces that β_* 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 combinator can be encoded in the λ-calculus with recursive types (or in the untyped λ-calculus). However, we shall take the view that it is too demanding to always forbid such use of the communication capabilities and we thus depart from [17].

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 a constraint $(\{t_i\ \text{CHAN}\} \subseteq \beta_*)$ is replaced by $\{(\gamma \subseteq \beta_*) \mid \gamma \in FV(t_i)\}$. Even if β_* occurs in one of $t_1, \cdots, t_n$ one could still let S_3' map β_* 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 [10] 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

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

With this new form of behaviour we can define S_3' by mapping β_* 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 an annotated type and effect system that integrates polymorphism, subtyping and effects [11]. Although the development was performed for a fragment of Concurrent ML we believe it equally

applicable to 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 whereas the issue of completeness as well as of complexity is still open.

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. We would like to thank the anonymous referees for their thought-provoking comments on the preliminary version of this work.

References

1. T. Amtoft, F. Nielson, H.R. Nielson, J. Ammann: Polymorphic subtypes for effect analysis: the dynamic semantics. This volume of SLNCS, 1997.
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. F. Henglein and C. Mossin: Polymorphic binding-time analysis. In *Proc. ESOP '94*, pages 287–301. SLNCS 788, 1994.
5. M.P. Jones: A theory of qualified types. In *Proc. ESOP '92*, pages 287–306. SLNCS 582, 1992.
6. P. Jouvelot and D.K. Gifford: Algebraic reconstruction of types and effects. In *Proc. POPL'91*, pages 303–310. ACM Press, 1991.
7. R. Milner: A theory of type polymorphism in programming. *Journal of Computer Systems*, 17:348–375, 1978.
8. J.C. Mitchell: Type inference with simple subtypes. *Journal of Functional Programming*, 1(3), 1991.
9. F. Nielson and H.R. Nielson: Constraints for polymorphic behaviours for Concurrent ML. In *Proc. CCL'94*. SLNCS 845, 1994.
10. H.R. Nielson and F. Nielson: Higher-order concurrent programs with finite communication topology. In *Proc. POPL'94*, pages 84–97. ACM Press, 1994.
11. H.R. Nielson, F. Nielson, T. Amtoft: Polymorphic subtypes for effect analysis: the static semantics. This volume of SLNCS, 1997.
12. 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.
13. J.H. Siekmann: Unification theory. *J. Symbolic Computation*, 7:207–274, 1989.
14. G.S. Smith: Polymorphic type inference for languages with overloading and subtyping. Ph.D thesis from Cornell, 1991.
15. G.S. Smith: Polymorphic inference with overloading and subtyping. In SLNCS 668, *Proc. TAPSOFT '93*, 1993. Also see: Principal type schemes for functional programs with overloading and subtyping: *Science of Computer Programming* 23, pp. 197-226, 1994.

16. J.P. Talpin and P. Jouvelot: Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3), pages 245–271, 1992.
17. J.P. Talpin and P. Jouvelot: The type and effect discipline. *Information and Computation*, 111, 1994.
18. Y.-M. Tang: Control flow analysis by effect systems and abstract interpretation. PhD thesis, Ecoles des Mines de Paris, 1994.
19. M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.
20. M. Tofte and L. Birkedal: Region-annotated types and type schemes, 1996. Submitted for publication.
21. A.K. Wright: Typing references by effect inference. In *Proc. ESOP '92*, pages 473–491. SLNCS 582, 1992.

A Proofs of Main Results

Algorithm $\mathcal{R}$

Lemma 26 Suppose $A \vdash (C', t', b') \longrightarrow (C'', t'', b'')$ and let $\gamma_1, \gamma_2 \in FV(C'')$. Then $C' \vdash \gamma_1 \Leftarrow^* \gamma_2$ holds iff $C'' \vdash \gamma_1 \Leftarrow^* \gamma_2$ 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 straightforward consequence of the assumption $C \vdash \gamma' \Leftarrow^* \gamma$. For (cycle), (shrink) and (boost) the "only if"-part follows from Fact 17: if $C' \vdash \gamma_1 \Leftarrow^* \gamma_2$ then $S\,C' \vdash S\,\gamma_1 \Leftarrow^* S\,\gamma_2$ and as $\gamma_1, \gamma_2 \notin Dom(S)$ this amounts to $S\,C' \vdash \gamma_1 \Leftarrow^* \gamma_2$ which is clearly equivalent to $C'' \vdash \gamma_1 \Leftarrow^* \gamma_2$.

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

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

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

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

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

Proposition 28 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 atomic. Then there exists (C_1', t_1', b_1') and (C_2', t_2', b_2'), which are equal up to renaming, such that

$A \vdash (C_1, t_1, b_1) \longrightarrow^{\leq 1} (C_1', t_1', b_1')$ 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

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

It will suffice to show that

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

for if e.g. $C \vdash \gamma_1' \Leftarrow^* \gamma_1$ holds then by (2) also $C \vdash \gamma_2' \Leftarrow^* \gamma_2$ 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

$C \vdash \gamma_1' \Leftarrow^* \gamma_2'$ and $C \vdash \gamma_2 \Leftarrow^* \gamma_1$ and
$C \vdash \gamma_2' \Leftarrow^* \gamma_1'$ and $C \vdash \gamma_1 \Leftarrow^* \gamma_2$

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

(redund) and (shrink) eliminating $(\gamma_1' \subseteq \gamma_1)$ and shrinking γ_2 into γ_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 $C \vdash \gamma_1' \Leftarrow^* \gamma_1$ 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

$C \cup \{\gamma_2' \subseteq \gamma_2\} \vdash \gamma_1' \Leftarrow^* \gamma_1$ and
$\gamma_2 \notin \mathit{FV}(\mathit{RHS}(C), A)$ and $\gamma_2 \neq \gamma_1$ and t, b, $\mathit{LHS}(C)$ is monotonic in γ_2.

Applying Fact 17 we get $S\,C \vdash S\,\gamma_1' \Leftarrow^* S\,\gamma_1$ 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 γ_2 into γ_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 $C \vdash \gamma_1 \Leftarrow^* \gamma_1'$ 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 γ_2.

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

(shrink) and (shrink) shrinking γ_1 into γ_1' and shrinking γ_2 into γ_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

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

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

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

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 $\mathit{LHS}(C)$ are monotonic in γ_1 as well as in γ_2: for then we for instance have $\{\gamma_1, \gamma_2\} \cap \mathit{NM}(t) = \emptyset$ and hence $\mathit{NM}(S_1\, t) = \mathit{NM}(S_2\, t) = \mathit{NM}(t)$.

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

(shrink) and (boost) shrinking γ_1 into γ_1' and boosting γ_2 into γ_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 γ). Then our assumption about the graph being cycle-free tells us that $\gamma_1' \neq \gamma_2'$, and the situation is

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

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

where (according to the side conditions for (shrink) and (boost)) it holds that $\gamma \notin \mathit{RHS}(C)$ and that t, b and each element in $\mathit{LHS}(C)$ is monotonic as well as anti-monotonic in γ. By Fact 21 and using that C is well-formed we infer that $\gamma \notin \mathit{FV}(C, t, b)$, 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.

$\underline{\gamma_1 \notin \{\gamma_2, \gamma_2', \gamma_1'\}}$ 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 $\gamma_1 \notin \mathit{FV}(\mathit{RHS}(C), A)$, where t and b and each element in $\mathit{LHS}(C)$ is monotonic in γ_1, where $\gamma_2 \notin \mathit{FV}(A)$, and where t and b and each element in $\mathit{LHS}(C)$ is anti-monotonic in γ_2.

As $\gamma_1 \neq \gamma_2'$ it is easy to see that $\gamma_1 \notin \mathit{FV}(\mathit{RHS}(S_2\, C), A)$ and that $S_2\, t$, $S_2\, b$ and $\mathit{LHS}(S_2\, C)$ is monotonic in γ_1; and as $\gamma_2 \neq \gamma_1'$ it is easy to see that $S_1\, t$, $S_1\, b$ and $\mathit{LHS}(S_1\, C)$ is anti-monotonic in γ_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 30 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. For "if" simply use rule (ins) with $S_0 = \mathit{Id}$. Next consider "only if" and write

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

so that $GEN(A,b)(C,t) = \forall(\vec{\gamma} : C_0).\ t$; this is well-formed by Fact 9.

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 $(C \setminus C_0) \cup RC_0$ by some S_0; simply take $S_0 = R$. Finally note that $\{\vec{\gamma}\} \cap FV((C \setminus C_0) \cup RC_0) = \emptyset$ 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 3 on the assumption) that

$$((C \setminus C_0) \cup RC_0) \cup C_0, A \vdash e\ :\ t \,\&\, b$$

and (gen) gives

$$(C \setminus C_0) \cup RC_0, A \vdash e\ :\ \forall(\vec{\gamma} : C_0).\ t \,\&\, b$$

and finally Lemma 2 gives the desired result:

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

This completes the proof. □

Theorem 31 If $\mathcal{W}(A,e) = (S,t,b,C)$ with A simple then $C, SA \vdash e\ :\ t \,\&\, b$.

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 $\mathcal{W}$. Notice that by Lemma 29 the condition for applying the induction hypothesis will always be fulfilled.

The case $e ::= c$**.** By Fact 8 it will be sufficient to show that

$$C, A \vdash c\ :\ t \,\&\, \emptyset \tag{11}$$

with $INST(\text{TypeOf}(c)) = (\text{Id}, t, \emptyset, C)$, as we can then use (sub) to arrive at the desired judgement

$$C, A \vdash c\ :\ \textit{Mk-simple}(t) \,\&\, \emptyset.$$

If TypeOf(c) is a type then this type equals t so (11) follows by an application of (con). Otherwise write TypeOf(c) $= \forall(\vec{\alpha_0}\vec{\beta_0} : C_0).\ t_0$ such that $t = Rt_0$ and $C = RC_0$ where R maps $\vec{\alpha_0}\vec{\beta_0}$ into fresh variables $\vec{\alpha'}\vec{\beta'}$; then (11) follows from the inference

$$\frac{RC_0, A \vdash c\ :\ \forall(\vec{\alpha_0}\vec{\beta_0} : C_0).\ t_0 \,\&\, \emptyset}{RC_0, A \vdash c\ :\ Rt_0 \,\&\, \emptyset}\ (\text{ins})$$

where the application of (ins) is justified since $RC_0 \vdash RC_0$.

The case $e ::= x$**.** If $A(x)$ is a type t_0 then the claim is that

$$\emptyset, A \vdash x \,:\, t_0 \,\&\, \emptyset$$

but this follows trivially by (id).

Otherwise write $A(x) = \forall(\vec{\alpha_0}\vec{\beta_0} : C_0).\ t_0$. The claim is that

$$R\,C_0, A \vdash x \,:\, R\,t_0 \,\&\, \emptyset$$

where R maps $\vec{\alpha_0}\vec{\beta_0}$ into fresh variables $\vec{\alpha'}\vec{\beta'}$. But this follows from the inference

$$\frac{R\,C_0, A \vdash x \,:\, \forall(\vec{\alpha_0}\vec{\beta_0} : C_0).\ t_0 \,\&\, \emptyset}{R\,C_0, A \vdash x \,:\, R\,t_0 \,\&\, \emptyset}\ \text{(ins)}$$

where the application of (ins) is justified since $R\,C_0 \vdash R\,C_0$.

The case $e ::= \texttt{fn}\ x \Rightarrow e_0$**.** 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

$$C, S\,(A[x : \alpha]) \vdash e_0 \,:\, t_0 \,\&\, b_0$$
$$C, (S\,A)[x : S\,\alpha] \vdash e_0 \,:\, t_0 \,\&\, \beta$$
$$C, S\,A \vdash \texttt{fn}\ x \Rightarrow e_0 \,:\, S\,\alpha \to^{\beta}\ t_0 \,\&\, \emptyset$$

using first Lemma 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 and 3 and then (sub) we get

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

Turning to e_2 the induction hypothesis gives

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

and using Lemma 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 $e ::= \texttt{let}\ x = e_1\ \texttt{in}\ e_2$**.** Concerning e_1 the induction hypothesis gives

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

and note that by Lemma 29 it holds that C_1 is atomic and hence well-formed. Next let $ts_1 = \mathit{GEN}(S_1\, A, b_1)(C_1, t_1)$ so that Lemmas 30, 2 and 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 3 we get

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

and hence using (let)

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

and this is the desired result.

The case $e ::= \texttt{rec}\ f\ x \Rightarrow e_0$**.** Concerning e_0 the induction hypothesis gives

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

Using Lemma 3, (sub), (abs) and (rec) we then get

$$C, (S\, A)[f : S\, \alpha_1 \rightarrow^{S\, \beta}\ S\, \alpha_2][x : S\, \alpha_1] \vdash e_0 \,:\, t_0 \,\&\, b_0$$

$$C, (S\, A)[f : S\, \alpha_1 \rightarrow^{S\, \beta}\ S\, \alpha_2][x : S\, \alpha_1] \vdash e_0 \,:\, S\, \alpha_2 \,\&\, S\, \beta$$

$$C, (S\, A)[f : S\, \alpha_1 \rightarrow^{S\, \beta}\ S\, \alpha_2] \vdash \texttt{fn}\ x \Rightarrow e_0 \,:\, S\, \alpha_1 \rightarrow^{S\, \beta}\ S\, \alpha_2 \,\&\, \emptyset$$

$$C, S\, A \vdash \texttt{rec}\ f\ x \Rightarrow e_0 \,:\, S\, \alpha_1 \rightarrow^{S\, \beta}\ S\, \alpha_2 \,\&\, \emptyset$$

which is the desired result.

The case $e ::= \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$**.** The induction hypothesis, Lemmas 2 and 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 $\mathcal{W}'(A, e) = (S_1, t_1, b_1, C_1)$ 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 14 ensures that $C_2 \vdash S_2\, C_1$. Using Lemmas 2 and 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 25 we get

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

which is the desired result. □