Polymorphic Types in Functional Logic Programming*

J.C. González-Moreno
M.T. Hortalá-González
M. Rodríguez-Artalejo

Dpto. Sistemas Informáticos y Programación
UCM, Madrid
jcmoreno@ei.uvigo.es
{teresa,mario}@sip.ucm.es

13 July 2001

Abstract

The rewriting logic CRWL has been proposed as a semantic framework for higher-order functional logic programming, using applicative rewriting systems as programs and lazy narrowing as the goal solving procedure. We present an extension of CRWL with a polymorphic type system, and we investigate the consequences of type discipline both at the semantic level and at the operational level. Semantically, models must be extended to incorporate a type universe. Operationally, lazy narrowing must maintain suitable type information in goals, in order to guarantee well-typed computed answers.

*This research has been partially supported by the Spanish National Project TIC98-0445-C03-02 "TREND" and the Esprit BRA Working Group EP-22457 "CCLII". A preliminary version of the paper, much shorter and including no proofs, was published as [20].

1 Introduction

Research on functional logic programming (FLP, for short) has been pursued for longer than ten years, aiming at the integration of the best features of functional programming (FP) and logic programming (LP). In this paper, we investigate the integration of two fundamental characteristics: *logical semantics*, mainly developed in the LP field, and *type discipline*, best understood in FP languages. Regarding the FP side, we restrict our attention to *non-strict* languages such as Haskell [43], whose advantages from the viewpoint of declarative programming are widely recognized. Thanks to lazy evaluation, functions in a non-strict language can sometimes return a result even if the values of some arguments are not known, or known only partially, and the semantic values intended for some expressions can be infinite objects. In the rest of the paper, "FP" must be understood as "non-strict FP".

Logical semantics characterizes the meaning of a pure logic program $\mathcal{P}$ (a set of definite Horn clauses) as its least Herbrand model, represented by the set of all the atomic formulas that are logical consequences of $\mathcal{P}$ in Horn logic [2, 14, 3]. Disregarding those proposals without a clear semantic basis, most early approaches to the integration of FP and LP, as e.g. [28, 16, 26] were based on the idea of adding equations to LP languages. This approach is appealing because equational logic is simple, well-known and widely applicable. Equational logic captures LP by representing Horn clauses as a particular kind of conditional equations, and it seems to capture also FP by viewing functional programs as sets of oriented equations, also known as *term rewriting systems* (shortly, TRSs) [10, 30, 6]. Certainly, term rewriting serves as an operational model for FP, but in spite of this equational logic does not provide a logical semantics for FP programs. In general, the set of all equations that are logical consequences of a functional program in equational logic *do not* characterize the meaning of the program. As a simple example, let $\mathcal{P}$ be the FP program consisting of the following two equations:

```
repeat1(X) ≈ [X|repeat1(X)]
repeat2(X) ≈ [X,X|repeat2(X)]
```

Here `[X|Xs]` represents a list with head `X` and tail `Xs`, and the notation `[X,X|repeat2(X)]` abbreviates `[X|[X|repeat2(X)]]`. In a non-strict FP language, it is understood that the expressions `repeat1(0)` and `repeat2(0)` have the same meaning, namely an infinite list formed by repeated occurrences of `0`. If equational logic would characterize the meaning of $\mathcal{P}$, both

expressions should be interchangeable for the purposes of equational deduction, which is not the case. In particular, `repeat1(0)` $\approx$ `repeat2(0)` cannot be deduced from $\mathcal{P}$ in equational logic.

In contrast to the failure of equational logic, *denotational semantics* [21] is able to characterize the meaning of functional programs. In our example, the common meaning of `repeat1(0)` and `repeat2(0)` is the limit of a sequence of partially defined lists with increasing information, built with the help of a special symbol $\bot$ which represents an undefined value:

$$\bot \sqsupseteq \texttt{[0|}\bot\texttt{]} \sqsupseteq \texttt{[0,0|}\bot\texttt{]} \sqsupseteq \texttt{[0,0,0|}\bot\texttt{]} \sqsupseteq \cdots$$

Borrowing ideas from denotational semantics, [19] proposed a rewriting logic CRWL to characterize the logical meaning of higher-order FP and FLP programs. The main results in [19] are existence of least Herbrand models for all programs, in analogy to the LP case, as well as soundness and completeness of a lazy narrowing calculus CLNC for goal solving. No type discipline was considered.

Regarding type discipline, most functional languages use Milner's type system [35, 9], which helps to avoid errors and to write more readable programs. This system has two crucial properties. Firstly, "well-typed programs don't go wrong", i.e., it is guaranteed that no type errors will occur during program execution, without any need of dynamic type checking at run time. Secondly, types are polymorphic, because they include type variables with a universal reading, standing for any type. For instance, a function to compute the length of a list admits the polymorphic type $\texttt{[}\alpha\texttt{]} \to \texttt{int}$, meaning that it will work for a list of values of any type α. Polymorphism promotes genericity of programs.

Type discipline in LP [12, 34] is not as well understood as in FP. Often one finds a distinction between the so-called *descriptive* and *prescriptive* views of types. The descriptive approach is applied to originally untyped programs, and views types *a posteriori* as approximations (usually regular supersets) of the success sets of predicates. On the contrary, the prescriptive approach views types *a priori* as imposing a restriction to the semantics of a program, so that predicates only accept arguments of the prescribed types. Usually, the prescriptive view leads to explicit type annotations in programs.

In our opinion, polymorphic type systems in Milner's style are also a good choice for LP and FLP languages. Types in Milner's type system have both a prescriptive and a descriptive rôle. More precisely, a type $\tau_1 \to \tau$ for a

function f *prescribes* an argument of type τ_1 and *describes* a result of type τ, which is guaranteed as long as the prescription for the argument is obeyed. The prescriptive part is in fact imposed, since an expression $f(e)$ is rejected at compile time if e has not type τ_1, in spite of the fact that $f(e)$ might happen to have a well defined value of type τ for some e that has not type τ_1. Mixing the prescriptive and descriptive views is also useful to understand some type systems for LP, where different types can be assigned to different modes of use of the same predicate. For each particular mode of use, one can think of a type prescription for the input arguments and a type description for the output arguments.

In the past, polymorphic type systems have been proposed for Prolog programs [39, 23, 22], for equational logic programs [24] and for higher-order logic programs in the language λ-Prolog [40, 41, 31]. Exactly as in Milner's original system, the aim of [39] was to guarantee that "well-typed programs don't go wrong" without any type checking at run time. On the contrary, the type systems in [23, 22, 24, 41, 31] can accept a wider class of well-typed programs, but type computations at run time are needed.

Currently, polymorphic type systems are supported by existing FLP languages such as Curry [13] and $\mathcal{TOY}$ [32]. In the case of $\mathcal{TOY}$, no dynamic type checking is performed by the current implementation. As a consequence, absence of type errors at run time is guaranteed for purely functional computations, but not for more complicated computations involving higher-order logic variables.

The present paper is a thoroughly revised, corrected and extended version of [20]. It is intended as a contribution to a better understanding of polymorphic type discipline in FLP languages with a logical semantics. Starting from the results in [19], we extend the rewriting logic CRWL and the narrowing calculus CLNC with a polymorphic type system, and we investigate the consequences of type discipline both at the semantic level and at the operational level. At the semantic level, we modify the models from [19] to incorporate a type universe. However, we do not require programs to be well-typed in order to be regarded as meaningful. Every program $\mathcal{P}$ has a logical meaning given by a least Herbrand model, which is well-behaved w.r.t. types in the case that $\mathcal{P}$ is well-typed. At the operational level, we modify CLNC to include some type information in goals. More precisely, a type environment assigning types to variables is maintained within each goal. The modified narrowing calculus is designed to be sound and complete in a reasonable sense w.r.t. the computation of well-typed solutions. Moreover, dynamic type checking

takes place only at those computation steps where some logic variable which is acting as a function becomes bound.

The rest of the paper is organized as follows. In Section 2 we introduce Higher-Order (shortly, HO) FLP programs as a special kind of applicative TRSs, as well as a polymorphic type system. In Section 3 we give all the constructions and results concerning the logical semantics of well-typed programs, including all the proofs that were omitted in [20], as well as new examples. In Section 4 we present the lazy narrowing calculus CLNC, showing by means of detailed examples that all the type checking mechanisms embodied in the calculus are really needed. The soundness and completeness of CLNC are presented in a revised formulation w.r.t. [20], and the proofs which were missing in [20] are now provided. All along the paper we include some succint comparison to related work. We summarize our conclusions in the final Section 5 and we collect most of the technical proofs in an Appendix.

2 Programming with Applicative Rewrite Systems

2.1 Types and Expressions

Since we are interested in HO FLP languages with a type discipline, we need a suitable syntax to represent types and expressions of any type. To introduce types, we assume a countable set $TVar$ of *type variables* α, β, ... and a countable ranked alphabet $TC = \bigcup_{n\in\mathbb{N}} TC^n$ of *type constructors* C. *Types* $\tau \in Type$ are built as

$$\tau \ ::= \ \alpha \ \ (\alpha \in TVar) \ \mid \ (C\,\tau_1 \ldots \tau_n) \ \ (C \in TC^n) \ \mid \ (\tau \to \tau')$$

Types without any occurrence of $\to$ are called *datatypes*. By convention, $C\,\overline{\tau}_n$ abbreviates $(C\,\tau_1 \ldots \tau_n)$, "$\to$" associates to the right, and $\overline{\tau}_n \to \tau$ abbreviates $\tau_1 \to \cdots \to \tau_n \to \tau$. The set of type variables occurring in τ is written $tvar(\tau)$. A type τ is called *monomorphic* iff $tvar(\tau) = \emptyset$, and *polymorphic* otherwise.

A *polymorphic signature* over TC is a triple $\Sigma = \langle TC, DC, FS\rangle$, where $DC = \bigcup_{n\in\mathbb{N}} DC^n$ resp. $FS = \bigcup_{n\in\mathbb{N}} FS^n$ are ranked sets of *data constructors* resp. *defined function symbols*. Moreover, each $c \in DC^n$ comes with a type declaration $c :: \overline{\tau}_n \to C\,\overline{\alpha}_k$, where $n, k \geq 0$, $\alpha_1, \ldots, \alpha_k$ are pairwise different,

τ_i are datatypes, and $tvar(\tau_i) \subseteq \{\alpha_1, \ldots, \alpha_k\}$ for all $1 \leq i \leq n$ (so-called *transparency property*). Also, every $f \in FS^n$ comes with a type declaration $f :: \overline{\tau}_n \to \tau$, where τ_i, τ are arbitrary types.

As we will see later on, functions $f \in FS^n$ must be defined by means of rewrite rules with n formal parameters. Moreover, the types declared in a signature for data constructors and defined functions are the *principal* or *most general* ones, which can be instantiated to more particular types (possibly involving "$\to$") for different uses of the constructor or function. Note that the principal types of data constructors must respect the transparency property and the additional restriction requiring the principal types of arguments to be datatypes. As we will see, transparency is essential for our results. The additional restriction is not imposed by current FP [43] or FLP [13, 32] languages, and we have not checked whether our results remain valid when dropping it. However, we believe that this restriction is not a severe limitation for practical programming; see Example 1 below.

In the sequel, we use the notation $(h :: \tau) \in_{var} \Sigma$ to indicate that Σ includes the type declaration $h :: \tau$ up to a renaming of type variables. For any signature Σ, we write $\Sigma_\perp$ for the result of extending Σ with a new data constructor $\perp :: \alpha$, intended to represent an undefined value that belongs to every type. As notational conventions, we use $c, d \in DC$, $f, g \in FS$ and $h \in DC \cup FS$, and we define the *arity* of $h \in DC^n \cup FS^n$ as $ar(h) = n$. In the sequel, we always suppose a given signature Σ, which will not always appear explicitly in our notation. Assuming a countable set $DVar$ of data *variables* $X, Y, \ldots$ (disjoint with $TVar$), *partial expressions* $e \in Exp_\perp$ are defined as follows:

$$e \ ::= \ X \ \ (X \in DVar) \ \mid \ \perp \ \mid \ h \ \ (h \in DC \cup FS) \ \mid \ (e\, e_1)$$

These expressions are usually called *applicative*, because $(e\ e_1)$ stands for the *application* operation (represented as juxtaposition) which applies the function denoted by e to the argument denoted by e_1. First-order (shortly, FO) expressions can be translated to applicative expressions by means of so-called *curried notation*. For instance, $f(X, g(Y))$ becomes $(f\ X\ (g\ Y))$. The set of data variables occurring in e is written $var(e)$. An expression e is called *closed* iff $var(e) = \emptyset$, and *open* otherwise. An expression e is called *linear* iff every $X \in var(e)$ has one single occurrence in e. Following a usual convention, we assume that application associates to the left, and we use the notation $e\, \overline{e}_n$ to abbreviate $e\, e_1 \ldots e_n$. Expressions $e \in Exp$ without

occurrences of $\bot$ are called *total*. Two important subclasses of expressions are *partial data terms* $t \in Term_\bot$, defined as

$$t \ ::= \ X \ \ (X \in DVar) \ \mid \ \bot \ \mid \ (c\,\overline{t}_n) \ \ (c \in DC^n)$$

and *partial patterns* $t \in Pat_\bot$, defined as:

$$\begin{array}{l} t \ ::= \ X \ \ (X \in DVar) \ \mid \ \bot \ \mid \ (c\,\overline{t}_m) \ \ (c \in DC^n, m \leq n) \ \mid \\ \qquad (f\,\overline{t}_m) \ \ (f \in FS^n, m < n) \end{array}$$

Note that expressions $(f\,\overline{t}_m)$ with $f \in FS^n$ and $m \geq n$ are not allowed as patterns, because they are potentially reducible using rewrite rules for f. *Total data terms* $t \in Term$ and *total patterns* $t \in Pat$ are defined analogously, but omitting $\bot$. Note that $Term_\bot \subset Pat_\bot \subset Exp_\bot$. As usual in FP, data terms are used to represent data values. Patterns generalize data terms and can be used as an intensional representation of functions, as we will see in Subsection 2.3.

Most functional languages allow also λ-*abstractions* of the form $\lambda X.e$ to represent functions. Some approaches to HO FLP also permit λ-abstractions, see e.g. [25, 46, 48, 33]. In spite of some known decidability results for particular cases, in general λ-abstractions can give rise to undecidable unification problems [17]. We restrict ourselves to λ-free applicative expressions, which are expressive enough for most programming purposes.

The next example illustrates some of the notions introduced so far.

Example 1 (Polymorphic Signature)
Let us consider a polymorphic signature with $TC^0 = \{\texttt{bool}, \texttt{nat}\}$, $TC^1 = \{\texttt{list}\}$, $TC^2 = \{\texttt{sum}\}$, $DC^0 = \{\texttt{true}, \texttt{false}, \texttt{z}, \texttt{nil}\}$, $DC^1 = \{\texttt{s}, \texttt{ls}, \texttt{rs}\}$, $DC^2 = \{\texttt{cons}\}$, $FS^0 = \{\texttt{one}\}$, $FS^1 = \{\texttt{not}, \texttt{negate}, \texttt{head}, \texttt{tail}, \texttt{unpack}, \texttt{wild}, \texttt{extend}, \texttt{pp}, \texttt{p}\}$, $FS^2 = \{\texttt{and}, \texttt{or}, \texttt{plus}, \texttt{map}, \texttt{snd}, \texttt{twice}, \texttt{(++)}\}$, $FS^3 = \{\texttt{third}, \texttt{split}\}$ and with the following type declarations:

$\texttt{true}, \texttt{false} :: \texttt{bool}$
$\texttt{cons} :: \alpha \to \texttt{list}\,\alpha \to \texttt{list}\,\alpha$
$\texttt{negate} :: \texttt{list}\,\texttt{bool} \to \texttt{list}\,\texttt{bool}$
$\texttt{tail}, \texttt{extend} :: \texttt{list}\,\alpha \to \texttt{list}\,\alpha$
$\texttt{and}, \texttt{or} :: \texttt{bool} \to \texttt{bool} \to \texttt{bool}$
$\texttt{plus} :: \texttt{nat} \to \texttt{nat} \to \texttt{nat}$
$\texttt{map} :: (\alpha \to \beta) \to \texttt{list}\,\alpha \to \texttt{list}\,\beta$
$\texttt{z}, \texttt{one} :: \texttt{nat}$
$\texttt{nil} :: \texttt{list}\,\alpha$
$\texttt{s} :: \texttt{nat} \to \texttt{nat}$
$\texttt{not} :: \texttt{bool} \to \texttt{bool}$
$\texttt{head} :: \texttt{list}\,\alpha \to \alpha$
$\texttt{p}, \texttt{pp} :: \texttt{nat} \to \texttt{bool}$
$\texttt{wild} :: \alpha \to \beta$

$$\texttt{(++)} :: \texttt{list}\,\alpha \to \texttt{list}\,\alpha \to \texttt{list}\,\alpha \qquad \texttt{snd} :: \alpha \to \beta \to \beta$$
$$\texttt{twice} :: (\alpha \to \alpha) \to \alpha \to \alpha \qquad \texttt{ls} :: \alpha \to \texttt{sum}\,\alpha\,\beta$$
$$\texttt{third} :: \alpha \to \beta \to \gamma \to \gamma \qquad \texttt{rs} :: \beta \to \texttt{sum}\,\alpha\,\beta$$
$$\texttt{split} :: \texttt{list}\,\alpha \to \texttt{list}\,\alpha \to \texttt{list}\,\alpha \to \texttt{bool}$$
$$\texttt{unpack} :: (\beta \to \beta) \to \alpha$$

Then, we can build *data terms* such as `(s X)`, `(cons (s X) (cons Y nil))`; *patterns* such as `(plus z)`, `(snd X)`, `(twice twice)`, `(twice (plus X))`; and *expressions*, like `(map (plus X) (cons (s X) nil))`, `(twice (plus X) Y)`. And we can also build the type `list (sum (`$\alpha \to$ `bool) `α`)`, for lists whose elements can represent either values of type α or boolean functions over such values.

In the sequel, we use Prolog notation for the list constructors, writing `[]` for `nil` and `[X|Xs]` for `cons X Xs`. We also write `[`α`]` for the type `list `α. In concrete examples, we sometimes use infix notation for other constructors and function symbols, writing e.g. `Xs ++ Ys` instead of `(++) Xs Ys`.

The following classification of expressions is useful: $X\,\overline{e}_m$, with $X \in DVar$ and $m \geq 0$, is called a *flexible expression*, while $h\,\overline{e}_m$ with $h \in DC \cup FS$ is called a *rigid expression*. Moreover, a rigid expression is called *active* iff $h \in FS$ and $m \geq ar(h)$, and *passive* otherwise. Note that any pattern is either a variable or a passive rigid expression. As we will see in Subsection 3.1, outermost reduction makes sense only for active rigid expressions.

Following the spirit of denotational semantics [21], we view $Pat_\perp$ as the set of finite elements of a semantic domain, and we define the *approximation ordering* $\sqsupseteq$ as the least partial ordering over $Pat_\perp$ satisfying the following properties: $\perp \sqsupseteq t$, for all $t \in Pat_\perp$; $X \sqsupseteq X$, for all $X \in DVar$; and $h\overline{t}_m \sqsupseteq h\overline{s}_m$ whenever these two expressions are patterns and $t_i \sqsupseteq s_i$ for all $1 \leq i \leq m$. $Pat_\perp$, and more generally any partially ordered set (shortly, poset), can be converted into a semantic domain by means of a technique called *ideal completion*; see e.g. [37]. Our semantics in Section 3 will be based on posets.

As usual, we define *type substitutions* $\sigma_t \in TSub$ as mappings $\sigma_t\colon TVar \to Type$ extended to $\sigma_t\colon Type \to Type$ in the natural way. Similarly, we consider *partial data substitutions* $\sigma_d \in DSub_\perp$ given by mappings $\sigma_d\colon DVar \to Pat_\perp$, *total data substitutions* $\sigma_d \in DSub$ given by mappings $\sigma_d\colon DVar \to Pat$, and *substitutions* given as pairs $\sigma = (\sigma_t, \sigma_d)$. By convention, we write $\tau\sigma_t$ instead of $\sigma_t(\tau)$, and $\theta_t\sigma_t$ for the composition of θ_t and σ_t, such that $\tau(\theta_t\sigma_t) = (\tau\theta_t)\sigma_t$ for any τ. We define the *domain* $dom(\sigma_t)$ as the set of all type variables α such that $\sigma_t(\alpha) \neq \alpha$, and the *range* $ran(\sigma_t)$ as $\bigcup_{\alpha \in dom(\sigma_t)} tvar(\sigma_t(\alpha))$. For any

subset $A \subseteq dom(\sigma_t)$ we define the *restriction* $\sigma_t \restriction A$ as the type substitution σ'_t such that $dom(\sigma'_t) = A$ and $\sigma'_t(\alpha) = \sigma_t(\alpha)$ for all $\alpha \in A$. Similar notions can be defined for data substitutions. the *identity substitution* $id = (id_t, id_d)$ is such that $id_t(\alpha) = \alpha$ for all $\alpha \in TVar$ and $id_d(X) = X$ for all $X \in DVar$.

The *subsumption ordering* over $Type$ is defined by the condition $\tau \leq \tau'$ iff $\tau' = \tau\sigma_t$ for some $\sigma_t \in TSub$. A similar ordering can be defined over $Exp_\perp$, and extended to work over $DSub_\perp$ by defining $\theta_d \leq \theta'_d$ iff $\theta'_d = \theta_d\sigma_d$ for some $\sigma_d \in DSub_\perp$. For any set of data variables $\mathcal{X}$, we use the notations $\theta_d \leq \theta'_d[\mathcal{X}]$ (resp. $\theta_d \leq \theta'_d[\backslash\mathcal{X}]$) to indicate that $X\theta'_d = X\theta_d\sigma_d$ holds for some $\sigma_d \in DSub_\perp$ and all $X \in \mathcal{X}$ (resp. all $X \notin \mathcal{X}$). The subsumption ordering over $Type$ also induces a subsumption ordering over $TSub$. Finally, let us mention the *approximation* ordering over $DSub_\perp$, defined by the condition $\sigma_d \sqsupseteq \sigma'_d$ iff $\sigma_d(X) \sqsupseteq \sigma'_d(X)$, for all $X \in DVar$.

2.2 Well-typed Expressions

Inspired by Milner's type system [35, 9] and by various approaches to type systems for LP [12, 34], we now introduce the notion of well-typed expression. We define a *type environment* as any set T of type assumptions $X :: \tau$ for data variables, such that T does not include two different assumptions for the same variable. The *domain* $dom(T)$ and the *range* $ran(T)$ of a type environment are the set of all data variables resp. type variables that occur in T. For any variable $X \in dom(T)$, the unique type τ such that $(X :: \tau) \in T$ is noted as $T(X)$. Given $\sigma_t \in TSub$, we define $T\sigma_t$ as the type environment T' such that $dom(T') = dom(T)$ and $T'(X) = T(X)\sigma_t$ for all $X \in dom(T)$. We write $T \leq T'$ iff $T' = T\sigma_t$ for some $\sigma_t \in TSub$. *Type judgements* $T \vdash_{WT} e :: \tau$ are derived by means of the following *type inference* rules:

VR $T \vdash_{WT} X :: \tau$, if $T(X) = \tau$.

ID $T \vdash_{WT} h :: \tau\sigma_t$, if $(h :: \tau) \in_{var} \Sigma_\perp$ and $\sigma_t \in TSub$.

AP $T \vdash_{WT} (e\ e_1) :: \tau$, if $T \vdash_{WT} e :: (\tau_1 \to \tau)$ and $T \vdash_{WT} e_1 :: \tau_1$, for some τ_1.

Note that the rule **ID** reflects the implicit universal quantification of type variables in the types of data constructors and defined functions. On the contrary, the rule **VR** treats the types of data variables as fixed by the

current type environment. This corresponds to the distinction between *let*-bound and λ-bound identifiers in the classical presentation of polymorphic type inference [35, 9].

An expression $e \in \mathit{Exp}_\perp$ is called *well-typed* in a type environment T iff there exists some type τ such that $T \vdash_{WT} e :: \tau$. Expressions that admit more than one type in T are called *polymorphic*. A well-typed expression always admits a so-called *principal type* that is more general than any other. Adapting ideas from [35, 9], we define a *type reconstruction* algorithm TR to compute principal types. Assume an expression e and a type environment T such that $\mathit{var}(e) \subseteq \mathit{dom}(T)$. Then $TR(T, e)$ returns a pair of the form (e^τ, E) where e^τ is a *type annotation* of e and E is a system of equations between types, expressing most general conditions for τ to be a valid type of e. The algorithm TR works by structural recursion on e:

VR $TR(T, X) = (X^\tau, \emptyset)$, if $T(X) = \tau$.

ID $TR(T, h) = (h^\tau, \emptyset)$, if $(h :: \tau) \in_{var} \Sigma_\perp$ is a fresh variant of h's type declaration.

AP $TR(T, (e\ e_1)) = ((e^{\tau_1 \to \gamma}\ e_1^{\tau_1})^\gamma, E \cup E_1 \cup \{\tau \approx \tau_1 \to \gamma\})$, if $TR(T, e) = (e^\tau, E)$, $TR(T, e_1) = (e_1^{\tau_1}, E_1)$, $\mathit{tvar}(E) \cap \mathit{tvar}(E_1) \subseteq \mathit{ran}(T)$, $\gamma \notin \mathit{tvar}(E) \cup \mathit{tvar}(E_1)$ is a fresh type variable.

Type-annotated expressions, as those returned by TR, have the following syntax:

$$\begin{array}{rcl} e^\tau & ::= & X^\tau \quad (X \in \mathit{DVar}, \tau \in \mathit{Type}) \ \mid \\ & & h^{\tau\sigma_t} \quad (h :: \tau \in_{var} \Sigma_\perp, \sigma_t \in \mathit{TSub}) \ \mid \ (e^{\tau_1 \to \tau}\ e_1^{\tau_1})^\tau \end{array}$$

Implicitly, we are assuming that a type-annotated expression never includes two different annotations for the same variable. In the sequel we often abbreviate type annotations by omitting some intermediate types. In particular, we write $(e^{\overline{\tau}_n \to \tau}\ \overline{e}_n^{\overline{\tau}_n})^\tau$, or even more simply $(e^{\overline{\tau}_n \to \tau}\ \overline{e}_n^{\overline{\tau}_n})$, to abbreviate a full type annotation of an expression of the form $(e\ \overline{e}_n)$. The following lemma says that type-annotated expressions correspond to the derivation of type judgements in a natural way. The straightforward proof is omitted.

Lemma 1 (Type Annotation vs. Type Derivation)
Given a type-annotated expression e^τ, let e be the expression obtained from e^τ by erasing all type annotations, and let T be the implicit type environment

of e^τ, which consists of all the type assumptions $X :: \tau$ such that X^τ occurs as a part of e^τ. Then $T \vdash_{WT} e :: \tau$. Reciprocally, whenever $T \vdash_{WT} e :: \tau$, there is some type annotation e^τ of e whose implicit type environment is T. □

The algorithm TR also returns a system E of equations between types. By definition, its set of *solutions* $TSol(E)$ consists of all $\sigma_t \in TSub$ such that $\tau\sigma_t = \tau'\sigma_t$ for all $\tau \approx \tau' \in E$. If E is *solvable* (i.e., $TSol(E) \neq \emptyset$), a most general solution $mgu(E) = \sigma_t \in TSol(E)$ can be computed by means of Robinson's unification algorithm (see e.g. [2]). The key properties of the type reconstruction algorithm are given by the next theorem, whose proof can be found in the Appendix.

Theorem 1 (Type Reconstruction)
Assume $TR(T, e) = (e^\tau, E)$. Then:

1. *For every $\sigma_t \in TSol(E)$: $T\sigma_t \vdash_{WT} e :: \tau\sigma_t$.*
2. *Reciprocally: if $T \leq T'$ and $\tau' \in Type$ are such that $T' \vdash_{WT} e :: \tau'$, then there exists $\sigma_t \in TSol(E)$ such that $T\sigma_t = T'$ and $e^\tau\sigma_t = e^{\tau'}$.* □

Assuming $TR(T, e) = (e^\tau, E)$ with solvable E, and $\sigma_t = mgu(E)$, we write $PT(T, e) = \tau\sigma_t$ for the *principal type* of e w.r.t. T, and $PA(T, e) = e^\tau\sigma_t$ for the *principal type annotation* of e w.r.t. T. Here, $e^\tau\sigma_t$ is meant as the result of applying σ_t to all the type annotations occurring within e^τ. In particular, the outermost type annotation in $e^\tau\sigma_t$ is $\tau\sigma_t$.

In order to compute a principal type for an expression e with $var(e) = \{X_1, \ldots, X_n\}$ one can invoke $TR(T, e)$ with $T = \{X_1 :: \alpha_1, \ldots, X_n :: \alpha_n\}$, where α_i are n pairwise different type variables. All the expressions from Example 1 are well-typed in suitable environments, and some of them have a polymorphic principal type. For instance, `twice twice` is well-typed in the empty environment. Moreover, $PA(\emptyset, \texttt{twice twice})$ can be computed as

$$(\texttt{twice}^{((\alpha\to\alpha)\to\alpha\to\alpha)\to(\alpha\to\alpha)\to\alpha\to\alpha}\ \texttt{twice}^{(\alpha\to\alpha)\to\alpha\to\alpha})^{(\alpha\to\alpha)\to\alpha\to\alpha}$$

The following three lemmata state some technically useful properties of type inference. Proofs are given in the Appendix.

Lemma 2 (Typing Monotonicity)
Assume $t \in Pat_\perp$, $\tau \in Type$ and a type environment T such that $T \vdash_{WT} t :: \tau$. Then $T \vdash_{WT} t' :: \tau$ holds also for every pattern $t' \sqsupseteq t$. □

Lemma 3 (Well-typed Substitution)
Assume two type environments T_0, T_1, a partial expression e such that $T_0 \vdash_{WT} e :: \tau$, and a substitution $\sigma = (\sigma_t, \sigma_d)$ which is well-typed in the sense that $T_1 \vdash_{WT} X\sigma_d :: T_0(X)\sigma_t$ for all $X \in var(e)$. Then, it is also true that $T_1 \vdash_{WT} e\sigma_d :: \tau\sigma_t$. □

Lemma 4 (Type Instantiation)
Assume $T \vdash_{WT} e :: \tau$. Then $T\sigma_t \vdash_{WT} e :: \tau\sigma_t$ holds for any substitution $\sigma_t \in TSub$. □

In fact, Lemma 4 is a particular case of Lemma 3. Moreover, Lemma 3 admits the following reformulation, which is helpful when working type annotations. A proof is also given in the Appendix.

Lemma 5 (Well-typed Substitution in a Type-annotated Expression)
Assume a type-annotated expression e^τ and a substitution $\sigma = (\sigma_t, \sigma_d)$ which is well-typed in the sense that $X\sigma_d$ can be annotated with type $\tau_0\sigma_t$ for every type-annotated variable X^{τ_0} occurring in e^τ. Then, $e^\tau\sigma$ is a type-annotated expression. □

When applying Lemma 5 in the sequel, we often assume that $e^\tau\sigma$ has been built as a principal type annotation.

As part of the definition of polymorphic signatures we have required a transparency property for the principal types of data constructors. Due to transparency, the types of the variables occurring in a data term t can be deduced from the type of t. It is useful to isolate those patterns that have a similar property. To this purpose, we define *transparent patterns* as

$$\begin{array}{rcl} t & ::= & X \;\; (X \in DVar) \;\mid\; \bot \;\mid\; (c\,\overline{t}_m) \;\; (c \in DC^n, m \leq n) \;\mid \\ & & (f\,\overline{t}_m) \;\; (f \in FS^n,\; m < n) \end{array}$$

where the subpatterns t_i in $(c\overline{t}_m)$ and $(f\,\overline{t}_m)$ must be recursively transparent, and the principal type of the defined function f in $(f\,\overline{t}_m)$ must be of the form $\overline{\tau}_m \to \tau$ with $tvar(\overline{\tau}_m) \subseteq tvar(\tau)$.

In the sequel, we say that a type which can be written as $\overline{\tau}_m \to \tau$ fulfilling $tvar(\overline{\tau}_m) \subseteq tvar(\tau)$ is *m-transparent*, and a function symbol f will be called m-transparent iff its principal type is m-transparent. Note that a data constructor c is always m-transparent for all $m \leq ar(c)$.

Types and patterns that are not transparent are called *opaque*. As a typical example of an opaque pattern, consider `(snd X)`, whose principal type $(\beta \to \beta)$ reveals no information on the type of `X`. Different instances of `(snd X)` actually keep the principal type $(\beta \to \beta)$, independently of the type of the expression substituted for `X`. The following *transparency lemmata* (proved in the Appendix) show that such a behaviour is not possible for transparent patterns.

Lemma 6 (Transparency)
Assume a transparent pattern t and two type environments T_1, T_2 such that $T_1 \vdash_{WT} t :: \tau$ and $T_2 \vdash_{WT} t :: \tau$, for a common type τ. Then $T_1(X) = T_2(X)$ holds for every $X \in var(t)$. □

In the sequel, we sometimes use the notation $T \vdash_{WT} a :: \tau :: b$ to indicate that the expressions a and b have a common type τ in the type environment T.

Lemma 7 (Transparent Decomposition)
Assume a m-transparent $h \in DC \cup FS$ such that $T \vdash_{WT} h\overline{a}_m :: \tau :: h\overline{b}_m$ holds for a common type τ. Then, there exists types τ_i such that $T \vdash_{WT} a_i :: \tau_i :: b_i$ holds for all $1 \leq i \leq m$. □

2.3 Programs

Following [19], we define CRWL programs as a special kind of applicative TRSs, but now requiring well-typedness. More precisely, assuming $f \in FS^n$ whose declared type (up to renaming of type variables) is of the form $f :: \overline{\tau}_n \to \tau$, a *well-typed* defining rule for f must have the following form:

$$\underbrace{f\ t_1\ \ldots\ t_n}_{\text{left hand side } (l)} \quad \to \quad \underbrace{r}_{\text{right hand side}} \quad \Leftarrow \quad \underbrace{C}_{\text{condition}} \quad \square \quad \underbrace{T}_{\text{type environment}}$$

where l must be linear, t_i must be transparent patterns, r must be an expression such that $var(r) \subseteq var(l)$, the condition C must consist of finitely many (possibly zero) so-called *joinability statements* $a \bowtie b$ where a, b are expressions, and T must be a type environment whose domain is the set of all data variables occurring in the rewrite rule, and such that: $T \vdash_{WT} t_i :: \tau_i$ for $1 \leq i \leq n$, $T \vdash_{WT} r :: \tau$ and $T \vdash_{WT} C ::$ `bool`. The symbol $\bot$ never occurs in a defining rule. A *well-typed* CRWL program can be any set of *well-typed*

defining rules for different symbols $f \in FS$. Neither termination nor confluence is required. In particular, the lack of confluence means that CRWL programs can define non-deterministic functions, whose usefulness for FLP languages has been advocated in [18]. Hence, the restriction $var(r) \subseteq var(l)$ is not motivated by confluence, but needed for a type preservation result presented as Theorem 2 in Subsection 3.1.

The meaning of joinability statements will be explained in the next section. An additional explanation is needed to understand the previous definitions. In $T \vdash_{WT} C :: \texttt{bool}$, we view C as an "expression" built from symbols in the current signature Σ, plus the two additional "operations" $(\bowtie) :: \alpha \to \alpha \to \texttt{bool}$ and $(\texttt{,}) :: \texttt{bool} \to \texttt{bool} \to \texttt{bool}$, used in infix notation to build conditions.

Note that defining rules in a well-typed program are *type-general* and *transparent*, because they match exactly the principal type declared for the corresponding function and they use transparent patterns in their left-hand sides. The transparency of function definitions made no sense in [19], but it is important in a typed setting. We consider also *untyped* programs, which are sets of untyped defining rules where the type environment T is missing. The restriction $var(r) \subseteq var(l)$ is not needed for untyped programs.

In practice, users of FP languages such as Haskell [43] or FLP languages such as $\mathcal{TOY}$ [32] provide type declarations only for data constructors. In most practical cases, this allows an automatic inference of the principal types of defined function symbols, using a type reconstruction algorithm embedded into the language's implementation. Note, however, that our class of CRWL programs allows so-called *polymorphic recursion*. The typability problem is known to be undecidable for programs which use polymorphic recursion [29], and any implemented type reconstruction algorithm is bound to fail sometimes for such programs.

The next program, based on the signature from Example 1, will be useful as a basis for other examples in the rest of the paper. It consists of obviously well-typed definitions, except for the following three exceptions: the rewrite rule defining the function `wild`, which has extra variables in its right-hand side; the first rewrite rule defining the function `extend`, which is not type-general; and the rewrite rule defining the function `unpack`, which has an opaque pattern in its left-hand side.

Example 2 (CRWL Program)

`not` :: `bool` → `bool`

```
not false → true ⇐ ∅ □ ∅
not true → false ⇐ ∅ □ ∅

or :: bool → bool → bool
or true X → true ⇐ ∅ □ {X :: bool}
or false X → X ⇐ ∅ □ {X :: bool}

negate :: [bool] → [bool]
negate [] → [] ⇐ ∅ □ ∅
negate [X|Xs] → [not X|negate Xs] ⇐ ∅ □ {X :: bool, Xs :: [bool]}

extend :: [α] → [α]
extend [] → [z] ⇐ ∅ □ ∅
extend [X|Xs] → [X,X|Xs] ⇐ ∅ □ {X :: α, Xs :: [α]}

and :: bool → bool → bool
and true X → X ⇐ ∅ □ {X :: bool}
and false X → false ⇐ ∅ □ {X :: bool}

head :: [α] → α
head [X|Xs] → X ⇐ ∅ □ {X :: α, Xs :: [α]}

one :: nat
one → s z

tail :: [α] → [α]
tail [X|Xs] → Xs ⇐ ∅ □ {X :: α, Xs :: [α]}

plus :: nat → nat → nat
plus z Y → Y ⇐ ∅ □ {Y :: nat}
plus (s X) Y → s (plus X Y) ⇐ ∅ □ {X, Y :: nat}

unpack :: (β → β) → α
unpack (snd X) → X ⇐ ∅ □ {X :: α}

wild :: α → β
wild X → Y ⇐ ∅ □ {X :: α, Y :: β}
```

```
p :: nat → bool
p (s Y) → true ⇐ ∅ □ {Y :: nat}

pp :: nat → bool
pp (s X) → p X ⇐ ∅ □ {X :: nat}

map :: (α → β) → [α] → [β]
map F [ ] → [ ] ⇐ ∅ □ {F :: (α → β)}
map F [X|Xs] → [F X|map F Xs] ⇐ ∅ □
                                  {F :: (α → β), X :: α, Xs :: [α]}
snd :: α → β → β
snd X Y → Y ⇐ ∅ □ {X :: α, Y :: β}

twice :: (α → α) → α → α
twice F X → F (F X) ⇐ ∅ □ {F :: (α → α), X :: α}

(++) :: [α] → [α] → [α]
[ ] ++ Ys → Ys ⇐ ∅ □ {Ys :: [α]}
[X|Xs] ++ Ys → [X|Xs ++ Ys] ⇐ ∅ □ {X :: α, Xs, Ys :: [α]}

third :: α → β → γ → γ
third X Y Z → Z ⇐ ∅ □ {X :: α, Y :: β, Z :: γ}

split :: [α] → [α] → [α] → bool
split Xs Ys Zs → true ⇐ Xs ⋈ Ys ++ Zs □ {Xs, Ys, Zs :: [α]}
```

In the previous program, most patterns in the left-hand sides are data terms of FO type, except for the HO pattern `(snd X)` in function `unpack` and the HO variable `F` in functions `map` and `twice`. The usefulness of HO patterns as intensional representations of functions is better illustrated in the next example, where `circuit` is used as an alias for the type `bool → bool → bool → bool`. Functions of this type are intended to represent simple circuits which receive three boolean inputs and return a boolean output.

Example 3 (Use of HO Patterns: Simple Circuits)

```
x1, x2, x3 :: circuit
x1 X1 X2 X3 → X1 ⇐ ∅ □ {X1, X2, X3 :: bool}
x2 X1 X2 X3 → X2 ⇐ ∅ □ {X1, X2, X3 :: bool}
```

```
x3 X1 X2 X3 → X3 ⇐ ∅ □ {X1, X2, X3 :: bool}

notGate :: circuit → circuit
notGate C X1 X2 X3 → not (C X1 X2 X3) ⇐ ∅ □
                                    {C :: circuit, X1, X2, X3 :: bool}
andGate :: circuit → circuit → circuit
andGate C1 C2 X1 X2 X3 → and (C1 X1 X2 X3) (C2 X1 X2 X3) ⇐ ∅ □
                                {C1, C2 :: circuit, X1, X2, X3 :: bool}
orGate :: circuit → circuit → circuit
orGate C1 C2 X1 X2 X3 → or (C1 X1 X2 X3) (C2 X1 X2 X3) ⇐ ∅ □
                                {C1, C2 :: circuit, X1, X2, X3 :: bool}
size :: circuit → nat
size x1 → z
size x2 → z
size x3 → z
size (notGate C) → s (size C) ⇐ ∅ □ {C :: circuit}
size (andGate C1 C2) → plus (size C1) (size C2) ⇐ ∅ □
                                                {C1, C2 :: circuit}
size (orGate C1 C2) → plus (size C1) (size C2) ⇐ ∅ □
                                                {C1, C2 :: circuit}
```

This obviously well-typed program should be completed with the definition of function `plus`, as in Example 2. Functions `x1`, `x2` and `x3` represent the basic circuits which just copy one of the inputs to the output. More interestingly, the HO functions `notGate`, `andGate` and `orGate` take circuits as parameters and build new circuits, corresponding to the logical gates `NOT`, `AND` and `OR`. The function `size`, whose definition uses HO patterns for circuits, computes the number of gates of a given circuit. Imagine that we are interested in a circuit whose output coincides with the majority of its three inputs. Using λ-notation, a circuit with this desired behaviour can be written as follows:

$$\lambda \texttt{X1}.\lambda \texttt{X2}.\lambda \texttt{X3}.\texttt{(and (or (and X1 X3) X2) (or X1 X3))}$$

In our λ-free setting, an intensionally different circuit with the same behaviour can be represented as a pattern:

```
(andGate (orGate (andGate x1 x3) x2) (orGate x1 x3))
```

More generally, many patterns in the signature of this example are useful representations of circuits. The problem of finding a pattern that realizes a

circuit with a given logical behaviour has been discussed in detail in [47]. The solution can be easily written as a $\mathcal{TOY}$ program [32], using a *lazy generate-and-test* method that exploits the combination of lazy evaluation and non-deterministic functions. The use of HO patterns as formal parameters and/or computed results is not possible in Haskell [43].

3 A Rewriting Logic for Program Semantics

3.1 Rewriting Calculi

In [19], a rewriting logic was proposed to deduce from an untyped CRWL program $\mathcal{P}$ certain statements that characterize the meaning of $\mathcal{P}$. More precisely, two kinds of statements must be considered: *approximation statements* $e \to t$, meaning that $t \in Pat_\perp$ approximates the value of $e \in Exp_\perp$; and *joinability statements* $a \bowtie b$, meaning that $a \to t$, $b \to t$ holds for some *total* $t \in Pat$. The collection of all t such that $e \to t$ can be deduced from $\mathcal{P}$ leads to a logical characterization of e's meaning, as we will see in Subsection 3.2. On the other hand, joinability statements are needed for conditions in rewrite rules, as well as for goals (see Section 4). They do not behave as equations in equational logic, for two reasons: t is required to be a total pattern rather than an arbitrary expression, and it is not required to be unique. Requiring unicity of t would lead to a deterministic version of joinability, which has been used under the name *strict equality* in several FLP languages, as e.g. [15, 38].

Roughly, a deduction of $e \to t$ in CRWL corresponds to a finite sequence of rewrite steps going from e to t in the TRS $\mathcal{P} \cup \{X \to \perp\}$. Unfortunately, this simple idea does not work directly, because it leads to an inconvenient treatment of non-determinism (see [18, 47] for details). Therefore, two special rewriting calculi were proposed in [19] to formalize CRWL deducibility in a HO setting. The first one, called Basic Rewriting Calculus (BRC for short) expresses reflexivity, monotonicity and transitivity of CRWL-reductions in a natural way. Its definition is as follows.

Definition 1 (The Basic Rewriting Calculus BRC)

$$\textbf{BT}: \quad e \to \bot \qquad\qquad \textbf{RF}: \quad e \to e$$

$$\textbf{TR}: \quad \frac{e \to e' \quad e' \to e''}{e \to e''} \qquad\qquad \textbf{MN}: \quad \frac{e \to e' \quad e_1 \to e_1'}{(e\, e_1) \to (e'\, e_1')}$$

$$\textbf{R}: \quad \frac{C}{l \to r} \quad \text{if } (l \to r \Leftarrow C) \in [\mathcal{P}]_\bot$$

$$\textbf{J}: \quad \frac{a \to t \quad b \to t}{a \bowtie b} \quad \text{if } t \text{ is a total pattern}$$

The inference rule **R** above uses the set of (possibly partial) instances of rewrite rules from $\mathcal{P}$, that is defined as follows, ignoring the type environments:

$$[\mathcal{P}]_\bot = \{(l \to r \Leftarrow C)\sigma_d \mid (l \to r \Leftarrow C \;\square\; T) \in \mathcal{P}, \sigma_d \in \mathit{DSub}_\bot\}$$

Due to the rules **BT** and **R**, BRC is not equivalent to classical rewriting. The alternative Goal Oriented Rewriting Calculus (shortly GORC), given below, looks still more unnatural from the classical rewriting viewpoint. The motivation to introduce GORC is the top-down, goal-oriented format of GORC proofs. As we will see in Section 4, this feature provides a useful basis for the design of goal-solving calculi.

Definition 2 (The Goal-Oriented Rewriting Calculus GORC)

$$\textbf{BT}: \quad e \to \bot \qquad\qquad \textbf{RR}: \quad X \to X \quad \text{if } X \in \mathit{DVar}$$

$$\textbf{OR}: \quad \frac{e_1 \to t_1 \quad \cdots \quad e_n \to t_n \quad C \quad r\, a_1 \ldots a_m \to t}{f\, e_1 \ldots e_n\, a_1 \ldots a_m \to t}$$

if $t \not\equiv \bot$ is a pattern, $m \geq 0$, and $f\, t_1 \ldots t_n \to r \Leftarrow C \in [\mathcal{P}]_\bot$

$$\textbf{DC}: \quad \frac{e_1 \to t_1 \quad \cdots \quad e_m \to t_m}{h\, e_1 \ldots e_m \to h\, t_1 \ldots t_m} \quad \text{if } h\, t_1 \ldots t_m \text{ is a rigid pattern}$$

$$\textbf{J}: \quad \frac{a \to t \quad b \to t}{a \bowtie b} \quad \text{if } t \text{ is a total pattern}$$

BRC and GORC are essentially equivalent, as shown by the following result.

Proposition 1 (Rewriting Calculi Equivalence)
For any CRWL program $\mathcal{P}$, BRC and GORC derive the same approximation and joinability statements.

Proof idea: Reasoning by induction on the structure of proofs, it is possible to show that BRC proofs can be converted into GORC proofs and viceversa. We omit a detailed reasoning because it would be very similar to the proof of Proposition 4.1 in [18], which deals with the FO fragment of CRWL in an untyped setting. This reference includes also more motivation for the introduction of both rewriting calculi. □

In the sequel we refer mainly to the GORC calculus. We use the notation $\mathcal{P} \vdash_{GORC} \varphi$ to assert that the statement φ can be deduced from the program $\mathcal{P}$ using GORC. Next we show a simple example of a GORC proof, based on the program from Example 2 and deriving `snd (tail [X])` $\bowtie$ `snd (tail [Y])`.

Example 4 (A Simple GORC Proof)

1. `snd (tail [X])` $\bowtie$ `snd (tail [Y])` by **JN**, 2, 3
2. `snd (tail [X])` $\rightarrow$ `snd [ ]` by **DC**, 4
3. `snd (tail [Y])` $\rightarrow$ `snd [ ]` by **DC**, 5
4. `tail [X]` $\rightarrow$ `[ ]` by **OR**, 6, 8
5. `tail [Y]` $\rightarrow$ `[ ]` by **OR**, 7, 8
6. `[X]` $\rightarrow$ `[X|[ ]]` by **DC**, 8, 9
7. `[Y]` $\rightarrow$ `[Y|[ ]]` by **DC**, 8, 10
8. `[ ]` $\rightarrow$ `[ ]` by **DC**
9. `X` $\rightarrow$ `X` by **RR**
10. `Y` $\rightarrow$ `Y` by **RR**

The next definition will be useful later on.

Definition 3 (Structured GORC Proofs)
The structure of a GORC proof Π *for statement* φ *(in symbols,* $\Pi \rightsquigarrow \varphi$*) obeys*

to the following abstract syntax:

$$
\begin{array}{rcl}
(\Pi \rightsquigarrow \varphi) & ::= & \mathbf{BT} \rightsquigarrow e \to \bot \\
& \mid & \mathbf{RR} \rightsquigarrow X \to X \\
& \mid & (\Pi_1 \rightsquigarrow e_1 \to t_1 \,\&\, \cdots \,\&\, \Pi_n \rightsquigarrow e_n \to t_n \,\&\, \Pi' \rightsquigarrow C \,\& \\
& & \quad \Pi'' \rightsquigarrow r\, \overline{a}_m \to t) \; +(\mathbf{OR}) \rightsquigarrow f\, \overline{e}_n\, \overline{a}_m \to t \\
\multicolumn{3}{l}{\quad \textit{if} \;\; (f\, \overline{t}_n \to r \Leftarrow C) \in [\mathcal{P}]_\bot} \\
& \mid & (\Pi_1 \rightsquigarrow e_1 \to t_1 \,\&\, \cdots \,\&\, \Pi_m \rightsquigarrow e_m \to t_m) \; +(\mathbf{DC}) \\
& & \quad \rightsquigarrow h\, \overline{e}_m \to h\, \overline{t}_m \\
& \mid & (\Pi_1 \rightsquigarrow a \to t \,\&\, \Pi_2 \rightsquigarrow b \to t) \; +(\mathbf{JN}) \rightsquigarrow a \bowtie b
\end{array}
$$

When writing structured GORC proofs in the rest of the paper, we sometimes abbreviate them by omitting some parts which are unneeded or can be deduced from the context.

There is a natural relation between GORC deducibility and the approximation ordering $\sqsupseteq$ over $Pat_\bot$, as shown by the following lemma. We omit the straightforward proof, which proceeds by induction over the size of GORC proofs. A proof of item 3 for the FO fragment of CRWL can be found in [18], Lemma 4.1.

Lemma 8 (Basic Properties of GORC)
Assume a (not necessarily well-typed) CRWL program $\mathcal{P}$. The following properties hold:

1. *For any $t, t' \in Pat_\bot$: $\mathcal{P} \vdash_{GORC} t' \to t \iff t \sqsupseteq t'$.*
2. *For any $e \in Exp_\bot$, $t, t' \in Pat_\bot$: $\mathcal{P} \vdash_{GORC} e \to t'$ and $t \sqsupseteq t' \implies \mathcal{P} \vdash_{GORC} e \to t$.*
3. *For any $e \in Exp_\bot$, $t \in Pat_\bot$ and $\sigma_d, \sigma'_d \in DSub_\bot$ such that $\sigma_d \sqsupseteq \sigma'_d$: $\mathcal{P} \vdash_{GORC} e\sigma_d \to t \implies \mathcal{P} \vdash_{GORC} e\sigma'_d \to t$ with a proof of the same size and structure.* □

Let us now consider the relationship between GORC provability and the polymorphic type system introduced in Subsection 2.2. Note that $\mathcal{P} \vdash_{GORC} e \to t$ corresponds to a purely functional computation reducing e to t. Although GORC derivations do not perform any kind of type checking, they

do always preserve types in the case of a well-typed program. The following *subject reduction lemma*, proved in the Appendix, guarantees that a single **OR** step is type-preserving.

Lemma 9 (One Step Subject Reduction for GORC)
Assume that a well-typed program $\mathcal{P}$ includes a defining rule $f\ \overline{t}_n \rightarrow r \Leftarrow C\ \square\ T_0$ for a defined symbol with principal type $f :: \overline{\tau}_n \rightarrow \tau_0$. Suppose also a type environment T and a substitution $\sigma = (\sigma_t, \sigma_d)$ which verifies $T \vdash_{WT} t_i\sigma_d :: \tau_i\sigma_t$ for all $1 \leq i \leq n$. Then $T \vdash_{WT} r\sigma_d :: \tau_0\sigma_t$. □

The proof of Lemma 9 relies on the Lemmata 3, 4, and 6. Type-preservation for arbitrary GORC derivations is ensured by the next result, whose full proof is given in the Appendix.

Theorem 2 (Subject Reduction for GORC)
Consider a well-typed program $\mathcal{P}$. Assume $e \in Exp_\perp$, $\tau \in Type$ and a type environment T such that $T \vdash_{WT} e :: \tau$. Then, for every $t \in Pat_\perp$:

$$\mathcal{P} \vdash_{GORC} e \rightarrow t \implies T \vdash_{WT} t :: \tau.$$

Proof idea: Induction on the size of a given GORC proof for $\mathcal{P} \vdash_{GORC} e \rightarrow t$. A case distinction is needed according to the GORC inference rule applied at the last step. The **OR** case is the most interesting one; it is proved with the help of Lemma 9. □

When defining well-typed program rules in Subsection 2.3, we have required absence of extra variables in the right-hand sides, as well as type generality and transparency of the left-hand sides. All these requirements are needed for the subject reduction property to hold. This is shown by the next example, based on the program from Example 2.

Example 5 (Subject Reduction Failure)

1. $\{\mathtt{X} :: \alpha\} \vdash_{WT} \mathtt{wild\ X} :: \mathtt{bool}$ and $\mathcal{P} \vdash_{GORC} \mathtt{wild\ X} \rightarrow \mathtt{z}$, but $\{\mathtt{X} :: \alpha\} \not\vdash_{WT} \mathtt{z} :: \mathtt{bool}$.

2. $\emptyset \vdash_{WT} \mathtt{extend\ [\,]} :: \mathtt{[bool]}$ and $\mathcal{P} \vdash_{GORC} \mathtt{extend\ [\,]} \rightarrow \mathtt{[z]}$, but $\emptyset \not\vdash_{WT} \mathtt{[z]} :: \mathtt{[bool]}$.

3. $\emptyset \vdash_{WT} \mathtt{unpack\ (snd\ z)} :: \mathtt{bool}$ and $\mathcal{P} \vdash_{GORC} \mathtt{unpack\ (snd\ z)} \rightarrow \mathtt{z}$, but $\emptyset \not\vdash_{WT} \mathtt{z} :: \mathtt{bool}$.

Our result on the existence of well-typed least Herbrand models for well-typed programs (see Lemma 11 and Theorem 3 in Subsection 3.2) crucially depend on the subject reduction property, and thus on the absence of extra variables in right-hand sides. Both the GORC calculus and the construction of least Herbrand models are concerned only with the behaviour of rewriting computations. Goal solving in the logic programming sense is the task of the lazy narrowing calculus CLNC presented in Section 4. In contrast to GORC, computations in CLNC do not instantiate extra variables in an arbitrary way. Therefore, it might be the case that extra variables on right-hand sides are no obstacle for the well-typed behaviour of lazy narrowing. We have not investigated this problem.

Independently of the extra variables issue, the possible occurrence of HO logic variables in goals implies that no analogon of the subject reduction property can be found for goal solving in CLNC. As we will see, dynamic type checking is needed in order to face this difficulty. In Section 4 we also use GORC proofs to witness the correctness of solutions computed by CLNC. For this reason, the following notion is useful to investigate the construction of well-typed CLNC computations.

Definition 4 (Type-annotated GORC Proofs)
Type-annotated GORC proofs are built by extending GORC proofs with consistent type information. More precisely, we use the following abstract syntax to describe the structure of type-annotated GORC proofs:

$$
\begin{array}{l}
(\Pi \rightsquigarrow \varphi) \;::=\; \mathbf{BT} \rightsquigarrow e^{\tau} \to \bot^{\tau} \\
\quad \mid\; \mathbf{RR} \rightsquigarrow X^{\tau} \to X^{\tau} \\
\quad \mid\; (\Pi_1 \rightsquigarrow e_1^{\tau_1} \to t_1^{\tau_1} \;\&\; \cdots \;\&\; \Pi_n \rightsquigarrow e_n^{\tau_n} \to t_n^{\tau_n} \;\&\; \Pi' \rightsquigarrow C^{\mathtt{bool}} \;\&\; \\
\qquad \Pi'' \rightsquigarrow (r^{\overline{\mu}_m \to \tau}\, \overline{a}_m^{\overline{\mu}_m})^{\tau} \to t^{\tau}) \;+(\mathbf{OR}) \rightsquigarrow (f^{\overline{\tau}_n \to \overline{\mu}_m \to \tau}\, \overline{e}_n^{\overline{\tau}_n}\, \overline{a}_m^{\overline{\mu}_m})^{\tau} \to t^{\tau} \\
\quad\; \textit{if} \;\; (f^{\overline{\tau}_n \to \overline{\mu}_m \to \tau}\, \overline{t}_n^{\overline{\tau}_n})^{\overline{\mu}_m \to \tau} \to r^{\overline{\mu}_m \to \tau} \Leftarrow C^{\mathtt{bool}} \in [\mathcal{P}]_{\bot}^{TA} \\
\quad \mid\; (\Pi_1 \rightsquigarrow e_1^{\tau_1} \to t_1^{\tau_1} \;\&\; \cdots \;\&\; \Pi_m \rightsquigarrow e_m^{\tau_m} \to t_m^{\tau_m}) \;+(\mathbf{DC}) \\
\qquad\; \rightsquigarrow (h^{\overline{\tau}_m \to \tau}\, \overline{e}_m^{\overline{\tau}_m})^{\tau} \to (h^{\overline{\tau}_m \to \tau}\, \overline{t}_m^{\overline{\tau}_m})^{\tau} \\
\quad \mid\; (\Pi_1 \rightsquigarrow a^{\tau} \to t^{\tau} \;\&\; \Pi_2 \rightsquigarrow b^{\tau} \to t^{\tau}) \;+(\mathbf{JN}) \rightsquigarrow a^{\tau} \bowtie b^{\tau}
\end{array}
$$

The notation $[\mathcal{P}]_{\bot}^{TA}$ refers to the set of type-annotated (and possibly partial) instances of defining rules from the well-typed program $\mathcal{P}$. It can be defined in two stages:

- *Due to Lemma 1, each rule* $(f\ \overline{t}_n \to r \Leftarrow C \ \square\ T) \in \mathcal{P}$ *for a function with principal type* $f :: \overline{\tau}_n \to \tau$ *can be type-annotated to become* $(f^{\overline{\tau}_n \to \tau}\ \overline{t}_n^{\overline{\tau}_n})^{\tau} \to r^{\tau} \Leftarrow C^{\mathtt{bool}}$ *(with implicit type environment* T*). Let* $\mathcal{P}^{TA}$ *be the set of all possible, principal type annotations of rules from* $\mathcal{P}$*, built in this way.*

- *Define* $[\mathcal{P}]^{TA}_{\perp}$ *as the set of all the type-annotated rule instances* $(l^{\tau} \to r^{\tau} \Leftarrow C^{\mathtt{bool}})\theta$*, such that* $(l^{\tau} \to r^{\tau} \Leftarrow C^{\mathtt{bool}}) \in \mathcal{P}^{TA}$ *and* $\theta = (\theta_t, \theta_d)$ *is well-typed in the sense of Lemma 5.*

All the expressions occurring within a type-annotated GORC proof are supposed to be fully type-annotated in the way explained in Subsection 2.2. Obviously, type-annotated GORC proofs always represent a well-typed computation. Some GORC proofs, however, cannot be type-annotated. Considering again the well-typed program $\mathcal{P}$ from Example 2, we get:

Example 6 (Type Annotation of GORC Proofs)

(a) The GORC proof from Example 4 can be type-annotated. The type-annotated version of the first line of the proof (somewhat abbreviated) would look as follows:

$$\begin{array}{c} (\mathtt{snd}^{[\alpha]\to\beta\to\beta}\ (\mathtt{tail}^{[\alpha]\to[\alpha]}\ [\mathtt{X}^{\alpha}]^{[\alpha]})^{[\alpha]})^{\beta\to\beta} \\ \bowtie \\ (\mathtt{snd}^{[\alpha]\to\beta\to\beta}\ (\mathtt{tail}^{[\alpha]\to[\alpha]}\ [\mathtt{Y}^{\alpha}]^{[\alpha]})^{[\alpha]})^{\beta\to\beta} \end{array}$$

This is an instance of the principal type-annotation, chosen to enable a consistent type-annotation of all the other statements occurring in the proof. The principal type annotation, namely

$$\begin{array}{c} (\mathtt{snd}^{[\alpha_1]\to\beta\to\beta}\ (\mathtt{tail}^{[\alpha_1]\to[\alpha_1]}\ \ [\mathtt{X}^{\alpha_1}]^{[\alpha_1]})^{[\alpha_1]})^{\beta\to\beta} \\ \bowtie \\ (\mathtt{snd}^{[\alpha_2]\to\beta\to\beta}\ (\mathtt{tail}^{[\alpha_2]\to[\alpha_2]}\ \ [\mathtt{Y}^{\alpha_2}]^{[\alpha_2]})^{[\alpha_2]})^{\beta\to\beta} \end{array}$$

would not allow a consistent type annotation of the second and third lines of the proof. This is because the **JN** case in Definition 4 requires t^{τ} be the same type-annotated pattern in the two premises $a^{\tau} \to t^{\tau}$ and $b^{\tau} \to t^{\tau}$.

(b) On the other hand, the following GORC proof *cannot* be type-annotated, because the only possible type-annotations of the lines 1, 2 and 3 are not consistent with the requirements of Definition 4 in the **JN** case.

1.	`snd (tail [z])` $\bowtie$ `snd (tail [true])`	by **JN**, 2, 3
2.	`snd (tail [z])` $\rightarrow$ `snd []`	by **DC**, 4
3.	`snd (tail [true])` $\rightarrow$ `snd []`	by **DC**, 5
4.	`tail [z]` $\rightarrow$ `[]`	by **OR**, 6, 8
5.	`tail [true]` $\rightarrow$ `[]`	by **OR**, 7, 8
6.	`[z]` $\rightarrow$ `[z|[]]`	by **DC**, 9, 8
7.	`[true]` $\rightarrow$ `[true|[]]`	by **DC**, 10, 8
8.	`[]` $\rightarrow$ `[]`	by **DC**
9.	`z` $\rightarrow$ `z`	by **DC**
10.	`true` $\rightarrow$ `true`	by **DC**

3.2 Models

A logical semantics for CRWL programs has been presented in [19] for an untyped HO language, and in [5] for a typed FO language with algebraic datatypes. Here we combine both approaches, using models with a data universe and a type universe. Let us recall some notions about *posets*, that are needed for our data domains.

A *poset* with bottom $\bot$ is any set S partially ordered by $\sqsubseteq$, with least element $\bot$. $Def(S)$ denotes the set of all maximal elements $u \in S$, also called *totally defined*. Assume $X \subseteq S$. X is a *directed set* iff for all $u, v \in X$ there exists $w \in X$ such that $u, v \sqsubseteq w$. X is a *cone* iff $\bot \in X$ and X is downwards closed w.r.t. $\sqsubseteq$. X is an *ideal* iff X is a directed cone. We write $\mathcal{C}(S)$ (resp. $\mathcal{I}(S)$) for the set consisting of all the subsets $X \subseteq S$ which are cones (resp. ideals) of S. $\mathcal{I}(S)$ ordered by set inclusion $\subseteq$ is a poset with bottom $\{\bot\}$, called the *ideal completion* of S. Mapping each $u \in S$ into the *principal ideal* $\langle u \rangle = \{v \in S \mid v \sqsubseteq u\}$ gives an order preserving embedding. There is a natural correspondence between the ideal completions of posets and a popular class of semantic domains, called *algebraic cpos* [21]; more details are given in [37, 18]. For our needs in this paper, the following intuitions will suffice: the elements of a poset S represent finite approximations of data values; defined elements in $Def(S)$ represent totally computed finite values of deterministic expressions; ideals in $\mathcal{I}(S)$ represent possibly infinite

values of deterministic expressions; and cones in $\mathcal{C}(S)$ represent values of non-deterministic expressions.

In the sequel we overload the symbol $\bot$ to stand for the bottom element of any poset, as well as for the syntactic $\bot$ used in partial expressions. Assume two posets with bottom D and E. To model deterministic resp. non-deterministic functions, we define:

$$[D \to_n E] = \{f\colon D \to \mathcal{C}(E) \mid \forall u, u' \in D\colon (u \sqsupseteq u' \Rightarrow f(u) \subseteq f(u'))\}$$
$$[D \to_d E] = \{f \in [D \to_n E] \mid \forall u \in D\colon f(u) \in \mathcal{I}(E)\}$$

Given $f \in [D \to_n E]$ and $X \in \mathcal{C}(D)$, we write $f(X)$ to abbreviate $\bigcup_{u\in X} f(u)$. It is easily checked that $f(X) \in \mathcal{C}(E)$. This fact will be implicitly used in the sequel. As models for CRWL programs we use algebras of the following kind.

Definition 5 (Algebras)
For any signature Σ we consider algebras of the form

$$\mathcal{A} = \langle D^{\mathcal{A}}, T^{\mathcal{A}}, @^{\mathcal{A}}, \Rightarrow^{\mathcal{A}}, ::^{\mathcal{A}}, \{C^{\mathcal{A}}\}_{C\in TC}, \{c^{\mathcal{A}}\}_{c::\tau\in DC}, \{f^{\mathcal{A}}\}_{f::\tau\in FS}\rangle$$

such that the following conditions hold:

1. *The* data universe $D^{\mathcal{A}}$ *is a poset, whose elements are called* data intensions.

2. *The* type universe $T^{\mathcal{A}}$ *is a set, whose elements are called* type intensions.

3. *The* apply operation $@^{\mathcal{A}}$*, that will be used in infix notation, belongs to* $[D^{\mathcal{A}} \times D^{\mathcal{A}} \to_n D^{\mathcal{A}}]$ *and satisfies* $\bot @^{\mathcal{A}} v = \langle\bot\rangle$ *for all* $v \in D^{\mathcal{A}}$*. For given elements* $u_i \in D^{\mathcal{A}}$*, the notation "*$u_0 \,@^{\mathcal{A}}\, u_1 \,@^{\mathcal{A}} \cdots @^{\mathcal{A}}\, u_k$*" represents a cone defined as* $\langle u_0\rangle$*, if* $k = 0$*, and* $(u_0 \,@^{\mathcal{A}} \cdots @^{\mathcal{A}}\, u_{k-1}) \,@^{\mathcal{A}}\, u_k$*, if* $k > 0$*.*

4. $\Rightarrow^{\mathcal{A}}\colon T^{\mathcal{A}} \times T^{\mathcal{A}} \to T^{\mathcal{A}}$*, that will be used in infix notation, interprets the type constructor "*$\to$*".*

5. $::^{\mathcal{A}} \subseteq D^{\mathcal{A}} \times T^{\mathcal{A}}$ *interprets type membership. For each* $l \in T^{\mathcal{A}}$*, the extension of* l*, defined as* $\mathcal{E}^{\mathcal{A}}(l) = \{u \in D^{\mathcal{A}} \mid u ::^{\mathcal{A}} l\}$*, must be a cone in* $D^{\mathcal{A}}$*.*

6. *For each* $C \in TC^n$, $C^{\mathcal{A}}\colon (T^{\mathcal{A}})^n \to T^{\mathcal{A}}$ *(simply* $C^{\mathcal{A}} \in T^{\mathcal{A}}$ *if* $n = 0$*), interprets the type constructor* C*.*

7. *For each* $c \in DC^n$, $c^{\mathcal{A}} \in D^{\mathcal{A}}$*.*

8. *For each* $f \in FS^n$, $f^{\mathcal{A}} \in D^{\mathcal{A}}$ *if* $n > 0$*, and* $f^{\mathcal{A}} \in \mathcal{C}(D^{\mathcal{A}})$*, if* $n = 0$*.*

9. *For all* h, k *such that* $h \in DC^n$, $0 \leq k \leq n$ *or* $h \in FS^n, 0 \leq k < n$*, and for all* $u_1, \ldots, u_k \in D^{\mathcal{A}}$ *there is* $v \in D^{\mathcal{A}}$ *such that* $h^{\mathcal{A}} @^{\mathcal{A}} u_1 @^{\mathcal{A}} \cdots @^{\mathcal{A}} u_k = \langle v \rangle$*. Moreover, if all the* u_i *are maximal then* v *is also maximal.*

The elements of $D^{\mathcal{A}}$ are called *data intensions* because they can be interpreted as functions by means of the apply operation $@^{\mathcal{A}}$, but they are not "true functions". Analogously, the *type intensions* belonging to $T^{\mathcal{A}}$ are interpreted as cones of data intensions by means of the membership relation $::^{\mathcal{A}}$, but they are not "true sets". Thanks to intensional semantics, one can work in a HO language while avoiding undecidability problems, especially undecidability of HO unification [17]. Variants of this idea have occurred previously in some LP and equational LP languages, as e.g. in [7, 23, 22, 24]. The paper [7] presents an untyped HO LP language called HiLog, with intensional semantics. In [23, 22, 24] one finds models and polymorphic type systems for FO LP and equational LP, as well as a suggestion to simulate HO programming by means of data intensions and user-defined `apply` predicates. This goes back to a technique proposed by Warren [49]. In order to well-type the `apply` predicates, type-generality of predicate definitions has to be abandoned. Moreover, [23, 22, 24] require explicit type annotations in terms, so that untyped expressions and programs have no semantic meaning.

To explain item 9 in Definition 5, let us abbreviate $h^{\mathcal{A}} @^{\mathcal{A}} u_1 @^{\mathcal{A}} \cdots @^{\mathcal{A}} u_k$ as $h^{\mathcal{A}}\, u_1 \ldots u_k$. This stands for the result of applying $h^{\mathcal{A}}$ to k arguments. Therefore, item 9 requires that the application of a data constructors and the partial application of a defined function to a number of arguments less than its arity, must return a deterministic result, totally defined if the arguments are. Given $h \in DC^n \cup FS^n$, we say that $h^{\mathcal{A}}$ is *deterministic* iff $h^{\mathcal{A}}\, u_1 \ldots u_n$ is an ideal for all $u_1, \ldots, u_n \in D^{\mathcal{A}}$. According to item 9, this is always the case for $c^{\mathcal{A}}$, for any $c \in DC^n$.

The technical requirements of Definition 5 are motivated by the properties of a particular class of algebras, called *Herbrand Algebras*.

Definition 6 (Herbrand Algebras)
Assume a type environment T with $dom(T) = DVar$. An algebra $\mathcal{A}$ is called a Herbrand algebra *over T iff the following conditions are satisfied:*

1. $D^{\mathcal{A}} = Pat_{\perp}$, *and* $T^{\mathcal{A}} = Type$.
2. $\tau_1 \Rightarrow^{\mathcal{A}} \tau = \tau_1 \to \tau$, *and* $C^{\mathcal{A}}(\tau_1, \ldots, \tau_n) = C\ \tau_1 \ldots \tau_n$, *for all* $C \in TC^n$.
3. $t ::^{\mathcal{A}} \tau$ *iff* $T \vdash_{WT} t :: \tau$, *for all* $t \in Pat_{\perp}$, $\tau \in Type$.
4. $\perp @^{\mathcal{A}} t = X @^{\mathcal{A}} t = \langle \perp \rangle$, *for all* $t \in Pat_{\perp}$, $X \in DVar$.
5. $t @^{\mathcal{A}} t_1 = \langle (t\ t_1) \rangle$, *whenever* $(t\ t_1) \in Pat_{\perp}$.
6. $c^{\mathcal{A}} = c$, *for all* $c \in DC$.
7. $f^{\mathcal{A}} = f$, *for all* $f \in FS$ *such that* $ar(f) > 0$.

A valuation $\eta = (\eta_t, \eta_d)$ *over $\mathcal{A}$ is given by two mappings* $\eta_t \colon TVar \to T^{\mathcal{A}}$ *(the* type valuation*) and* $\eta_d \colon DVar \to D^{\mathcal{A}}$ *(the* data valuation*). A valuation η is* totally defined *iff* $\eta_d(X) \in Def(D^{\mathcal{A}})$ *for all* $X \in DVar$. *We write* $Val(A)$ *resp.* $DefVal(\mathcal{A})$ *for the set of all valuations resp. totally defined valuations over $\mathcal{A}$. The values of types and expressions in $\mathcal{A}$ under η are computed recursively:*

- $[\![\alpha]\!]^{\mathcal{A}}\eta = \eta_t(\alpha)$, *for* $\alpha \in Type$.
- $[\![(C\ \tau_1 \ldots \tau_n)]\!]^{\mathcal{A}}\eta = C^{\mathcal{A}}([\![\tau_1]\!]^{\mathcal{A}}\eta, \ldots, [\![\tau_n]\!]^{\mathcal{A}}\eta)$, *for* $C \in TC^n$.
- $[\![\tau_1 \to \tau]\!]^{\mathcal{A}}\eta = [\![\tau_1]\!]^{\mathcal{A}}\eta \Rightarrow^{\mathcal{A}} [\![\tau]\!]^{\mathcal{A}}\eta$.
- $[\![\perp]\!]^{\mathcal{A}}\eta = \langle \perp \rangle$.
- $[\![X]\!]^{\mathcal{A}}\eta = \langle \eta_d(X) \rangle$, *for* $X \in DVar$.
- $[\![c]\!]^{\mathcal{A}}\eta = \langle c^{\mathcal{A}} \rangle$, *for* $c \in DC$.
- $[\![f]\!]^{\mathcal{A}}\eta = \langle f^{\mathcal{A}} \rangle$ *if* $ar(f) > 0$ *and* $f^{\mathcal{A}}$ *otherwise, for* $f \in FS$.
- $[\![(e\ e_1)]\!]^{\mathcal{A}}\eta = [\![e]\!]^{\mathcal{A}}\eta @^{\mathcal{A}} [\![e_1]\!]^{\mathcal{A}}\eta$.

Sometimes we write $[\![e]\!]^{\mathcal{A}}\eta_d$ *and* $[\![\tau]\!]^{\mathcal{A}}\eta_t$ *for* $[\![e]\!]^{\mathcal{A}}\eta$ *and* $[\![\tau]\!]^{\mathcal{A}}\eta$, *respectively.*

Note that in the case of Herbrand algebras, valuations are the same as substitutions. Some simple properties of evaluation are stated in the next proposition, whose proof is sketched in the Appendix.

Proposition 2 (Basic Properties of Evaluation)
Assume $\tau \in Type$, $e \in Exp_\perp$, $t \in Pat_\perp$ and $\eta \in Val(\mathcal{A})$. Then:

1. $[\![\tau]\!]^{\mathcal{A}}\eta_t \in T^{\mathcal{A}}$, *and* $[\![e]\!]^{\mathcal{A}}\eta_d \in \mathcal{C}(D^{\mathcal{A}})$.
2. *If $f^{\mathcal{A}}$ is deterministic for every $f \in FS$ occurring in e, then* $[\![e]\!]^{\mathcal{A}}\eta_d \in \mathcal{I}(D^{\mathcal{A}})$.
3. $[\![t]\!]^{\mathcal{A}}\eta_d = \langle v \rangle$, *for some* $v \in D^{\mathcal{A}}$. *Moreover, if $t \in Pat$ is total and $\eta \in DefVal(\mathcal{A})$, then $v \in Def(D^{\mathcal{A}})$.*
4. *If $\mathcal{A}$ is a Herbrand algebra, then* $[\![\tau]\!]^{\mathcal{A}}\eta_t = \tau\eta_t$ *and* $[\![t]\!]^{\mathcal{A}}\eta_d = \langle t\eta_d \rangle$. □

Another basic result establishes a natural relationship between substitution and evaluation. We omit the straightforward proof by induction over the syntactic structure of τ, e. Similar results can be found in [18, 5].

Lemma 10 (Substitution Lemma)
Consider a valuation $\eta = (\eta_t, \eta_d) \in Val(\mathcal{A})$ and a substitution $\sigma = (\sigma_t, \sigma_d)$. Define another valuation $\eta\sigma = (\eta\sigma_t, \eta\sigma_d) \in Val(\mathcal{A})$ by the conditions:

- $(\eta\sigma_t)(\alpha) = [\![\sigma_t(\alpha)]\!]^{\mathcal{A}}\eta_t$ *for all $\alpha \in TVar$.*
- $(\eta\sigma_d)(X) = v \in D^{\mathcal{A}}$ *such that* $[\![\sigma_d(X)]\!]^{\mathcal{A}}\eta_d = \langle v \rangle$ *for all $X \in DVar$; note that v exists because of Proposition 2, item 3.*

Then, $[\![\tau\sigma_t]\!]^{\mathcal{A}}\eta_t = [\![\tau]\!]^{\mathcal{A}}\eta\sigma_t$ *for every $\tau \in Type$, and* $[\![e\sigma_d]\!]^{\mathcal{A}}\eta_d = [\![e]\!]^{\mathcal{A}}\eta\sigma_d$ *for every $e \in Exp_\perp$.* □

Algebras and valuations are not always well-behaved w.r.t. types. The following definition isolates the well-typed ones.

Definition 7 (Well-typed Algebras)

1. *A valuation $\eta = (\eta_t, \eta_d)$ over $\mathcal{A}$ is well-typed w.r.t. a type environment T iff every $X :: \tau \in T$ verifies that $\eta_d(X) \in \mathcal{E}^{\mathcal{A}}([\![\tau]\!]^{\mathcal{A}}\eta_t)$.*
2. *An algebra $\mathcal{A}$ is well-typed iff for every type valuation η_t:*

(a) $c^{\mathcal{A}} \in \mathcal{E}^{\mathcal{A}}([\![\tau]\!]^{\mathcal{A}}\eta_t)$, *for* $c :: \tau \in DC^n$.

(b) $f^{\mathcal{A}} \in \mathcal{E}^{\mathcal{A}}([\![\tau]\!]^{\mathcal{A}}\eta_t)$, *for* $f :: \tau \in FS^n$, $n > 0$.

(c) $f^{\mathcal{A}} \subseteq \mathcal{E}^{\mathcal{A}}([\![\tau]\!]^{\mathcal{A}}\eta_t)$, *for* $f :: \tau \in FS^0$.

(d) *For all* $u, u_1 \in D^{\mathcal{A}}$, *for all* $t, t_1 \in T^{\mathcal{A}}$: *if* $u \in \mathcal{E}^{\mathcal{A}}(t_1 \Rightarrow^{\mathcal{A}} t)$ *and* $u_1 \in \mathcal{E}^{\mathcal{A}}(t_1)$ *then* $u \,@^{\mathcal{A}}\, u_1 \subseteq \mathcal{E}^{\mathcal{A}}(t)$.

The next result, proved in the Appendix, guarantees that the evaluation of well-typed expressions in well-typed algebras behaves as expected.

Proposition 3 (Well-typed Evaluation)
Assume a type environment T, *a well-typed algebra* $\mathcal{A}$, *and a valuation* $\eta \in \mathit{Val}(\mathcal{A})$ *that is well-typed w.r.t.* T. *Then, for every* $e \in \mathit{Exp}_\perp$ *such that* $T \vdash_{WT} e :: \tau$, *one has* $[\![e]\!]^{\mathcal{A}}\eta_d \subseteq \mathcal{E}^{\mathcal{A}}([\![\tau]\!]^{\mathcal{A}}\eta_t)$. □

In the rest of this section we prove the existence of least Herbrand models for CRWL programs. The notion of model, borrowed from [19], does not depend on the type system.

Definition 8 (Models)
Given an algebra $\mathcal{A}$ *and a program* $\mathcal{P}$, *we define:*

1. $\mathcal{A}$ *satisfies an approximation statement* $e \to t$ *under* $\eta \in \mathit{Val}(\mathcal{A})$ *iff* $[\![e]\!]^{\mathcal{A}}\eta \supseteq [\![t]\!]^{\mathcal{A}}\eta$.

2. $\mathcal{A}$ *satisfies a condition* C *under* $\eta \in \mathit{Val}(\mathcal{A})$ *iff for every* $a \bowtie b \in C$ *there is some totally defined element* $v \in [\![a]\!]^{\mathcal{A}}\eta \cap [\![b]\!]^{\mathcal{A}}\eta$.

3. $\mathcal{A}$ *satisfies a rewrite rule* $(l \to r \Leftarrow C \;\square\; T)$ *iff* $[\![l]\!]^{\mathcal{A}}\eta \supseteq [\![r]\!]^{\mathcal{A}}\eta$ *holds for every* $\eta \in \mathit{Val}(\mathcal{A})$ *such that* $\mathcal{A}$ *satisfies* C *under* η.

4. $\mathcal{A}$ *is a model of* $\mathcal{P}$ *iff* $\mathcal{A}$ *satisfies all the rewrite rules belonging to* $\mathcal{P}$.

Note that neither η_t nor the type environments attached to rewrite rules are needed in the previous definition. Therefore, untyped programs can also have models. In the sequel, we write $(\mathcal{A}, \eta) \vDash \varphi$ (where φ maybe an approximation statement or a joinability statement) to note that $\mathcal{A}$ satisfies φ under η, and we write $\mathcal{A} \vDash \mathcal{P}$ to note that $\mathcal{A}$ is a model of $\mathcal{P}$. We are mainly interested in *least Herbrand models*, that are defined as follows:

Definition 9 (Least Herbrand Models)
Assume a program $\mathcal{P}$ and a type environment T such that $dom(T) = DVar$. The least Herbrand model *of $\mathcal{P}$ over T, noted as $\mathcal{M}_{\mathcal{P}}(T)$, is the unique Herbrand algebra over T that satisfies the two following requirements:*

1. $f\,\overline{t}_{n-1}\,@^{\mathcal{M}_{\mathcal{P}}(T)}\,t_n = \{t \in Pat_\perp \mid \mathcal{P} \vdash_{GORC} f\,\overline{t}_n \to t\}$, *for* $f \in FS^n$, $n > 0$.
2. $f^{\mathcal{M}_{\mathcal{P}}(T)} = \{t \in Pat_\perp \mid \mathcal{P} \vdash_{GORC} f \to t\}$, *for* $f \in FS^0$.

By inspecting Definitions 5, 6 and 9, it can be checked that $\mathcal{M}_{\mathcal{P}}(T)$ is well defined, and such that the following condition holds for any $f \in FS^n$ and arbitrary patterns $t_1, \ldots, t_n \in Pat_\perp$: $f^{\mathcal{M}_{\mathcal{P}}(T)}\,\overline{t}_n = \{t \in Pat_\perp \mid \mathcal{P} \vdash_{GORC} f\,\overline{t}_n \to t\}$. More precisely, the essential properties of $\mathcal{M}_{\mathcal{P}}(T)$ are given by the next two lemmata, whose proofs can be found in the Appendix.

Lemma 11 (Least Herbrand Models are Well Defined)
Assume a CRWL program $\mathcal{P}$ and the type environment T such that $dom(T) = DVar$. Then:

1. *$\mathcal{M}_{\mathcal{P}}(T)$ is a well-defined Herbrand algebra, in the sense of Definitions 5, 6.*
2. *Moreover, if $\mathcal{P}$ is well-typed, then $\mathcal{M}_{\mathcal{P}}(T)$ is also well-typed.* □

Lemma 12 (Characterization Lemma)
Consider a least Herbrand model $\mathcal{M}_{\mathcal{P}}(T)$ and $\sigma_d \in DSub_\perp$, which serves as a valuation over $\mathcal{M}_{\mathcal{P}}(T)$. Then:

1. *For every $e \in Exp_\perp$: $[\![e]\!]^{\mathcal{M}_{\mathcal{P}}(T)}\sigma_d = \{t \in Pat_\perp \mid \mathcal{P} \vdash_{GORC} e\sigma_d \to t\}$*
2. *For every approximation or joinability statement φ:*

$$(\mathcal{M}_{\mathcal{P}}(T), \sigma_d) \vDash \varphi \iff \mathcal{P} \vdash_{GORC} \varphi\sigma_d.$$ □

According to item 2 from Lemma 11, $\mathcal{M}_{\mathcal{P}}(T)$ is well-typed whenever $\mathcal{P}$ is well-typed. The proof of this fact relies essentially on the subject reduction property of GORC, given by Theorem 2. According to Lemma 12, semantic validity in $\mathcal{M}_{\mathcal{P}}(T)$ behaves the same as derivability in GORC. This allows to prove the next theorem, which is the main result in this section.

Theorem 3 (Canonicity of Least Herbrand Models)
$\mathcal{M}_\mathcal{P}(T) \vDash \mathcal{P}$ holds for any program $\mathcal{P}$, and for every approximation or joinability statement φ the three following conditions are equivalent:

1. $\mathcal{P} \vdash_{GORC} \varphi$.
2. $(\mathcal{A}, \eta) \vDash \varphi$, *for all* $A \vDash \mathcal{P}$ *and all* $\eta \in DefVal(\mathcal{A})$.
3. $(\mathcal{M}_\mathcal{P}(T), id) \vDash \varphi$. □

A full proof of this theorem is given in the Appendix. The equivalence of items 1 and 2 shows that GORC is sound and complete for deriving those statements which hold in all models of a given program $\mathcal{P}$ under all possible *totally defined valuations*. The result cannot be weakened to arbitrary valuations, as shown by the statement $X \bowtie X$. Moreover, item 3 characterizes those statements in terms of truth in $\mathcal{M}_\mathcal{P}(T)$. Therefore, the canonic model $\mathcal{M}_\mathcal{P}(T)$ can be viewed as the logical semantics of $\mathcal{P}$, in analogy to the least Herbrand model of a LP program over a Herbrand universe with variables [14, 3]. Least Herbrand models of FO CRWL programs can be also characterized as least fixpoints and as free objects in a category of models [36, 18, 5]. These results could be extended to the present HO setting, but we will not dwell on this issue.

4 A Lazy Narrowing Calculus for Goal Solving

In this section we extend the untyped lazy narrowing calculus CLNC from [19] to prevent type errors at run time. Other existing narrowing calculi, as e.g. [42, 44, 45, 25, 48, 33], can be used as a basis for the goal solving mechanism of a HO FLP language. With the exception of [42], all these approaches deal with simply-typed λ-terms, which give rise to undecidable unification problems in the general case [17]. The calculus in [42] is designed for an untyped, λ-free setting, similarly to our former approach in [19]. The main novelty of the present approach is the treatment of polymorphic types in the context of a HO FLP language which avoids the complications related to general HO unification of λ-terms. As argued in [47], artificial incompleteness problems in narrowing calculi often arise due to an improper choice of the

semantics. This is not the case for our logical semantics from Section 3, which has been chosen to reflect the computational behaviour of lazy functions.

We recall that initial goals G for CLNC are finite systems of joinability statements $a \bowtie b$. A correct solution for such a G is any total data substitution θ_d such that $\mathcal{P} \vdash_{GORC} a\theta_d \bowtie b\theta_d$ for all $a \bowtie b \in G$. Soundness and completeness of CLNC, as proved for untyped programs in [19], remain valid for well-typed programs. In the typed setting, however, one would like to avoid the computation of ill-typed answers, whenever the program and the initial goal are well-typed. Unfortunately, there are problematic situations where run time type errors can occur in CLNC. A first kind of problem arises when a HO logic variable F occurs as head in some flexible expression $(F\,\overline{e}_m)$. In CLNC there are special transformations to guess a suitable pattern t as binding for F in such cases, and sometimes $t\,\overline{e}_m$ may become ill-typed. For instance, given the well-typed goal `F X` $\bowtie$ `true`, CLNC can guess the binding `F` $\mapsto$ `plus z`, leading to an ill-typed goal and eventually to the ill-typed solution {`F` $\mapsto$ `plus z, X` $\mapsto$ `true`}. As a second example, consider `map F [true,X]` $\bowtie$ `[Y,false]`. This is also a well-typed goal, but CLNC can compute an ill-typed solution represented by the set of bindings {`F` $\mapsto$ `plus z, X` $\mapsto$ `false, Y` $\mapsto$ `true`}.

A second kind of problematic situation is related to statements $h\,\overline{a}_m \bowtie h\,\overline{b}_m$, joining two *rigid and passive* expressions. In CLNC a *decomposition* transformation reduces such condition to a system of simpler conditions $a_i \bowtie b_i$. We say that such a decomposition step is *transparent* iff h is m-transparent and *opaque* otherwise. In the case of an opaque decomposition step, some of the new conditions $a_i \bowtie b_i$ may become ill-typed. Consider for instance the two well-typed goals `snd true` $\bowtie$ `snd z` and `snd (map s [])` $\bowtie$ `snd (map not [])`. Both of them become ill-typed after a decomposition step. In the first case the goal is unsolvable, while in the second case the computation ultimately succeeds, in spite of the type error. Opaque decomposition steps are also possible for approximation statements $h\,\overline{a}_m \rightarrow h\,\overline{t}_m$. Moreover, guessing a binding for a logic variable F which occurs as head in a flexible expression $(F\,\overline{e}_m)$, can subsequently give rise to opaque decomposition.

Unfortunately, avoiding opaque decomposition is not easy, because its eventual occurrence is undecidable, even for reasonably simple programs and goals. A proof of this result, as well as a brief discussion on the harmfulness of opaque decompositions from a practical viewpoint, can be found in Subsection 4.2 below. The other source of type errors described above, namely

the computation of ill-typed bindings logic variables, is a well-known problem. It can arise in the case of HO logic variables, and sometimes also in the case of FO logic variables (for programs which allow subtyping or do not impose transparency and type-generality requirements). Different solutions have been proposed, some of which are summarized below.

As shown in [23, 22, 24] for the case of FO typed SLD resolution and FO typed narrowing, respectively, a goal solving procedure that works with fully type-annotated goals can detect ill-typed bindings at run time, since they cause failure at the level of type unification. For instance, when trying to solve the type-annotated goal $(\mathtt{F}^{\alpha\to\mathtt{bool}}\,\mathtt{X}^{\alpha})^{\mathtt{bool}} \bowtie \mathtt{true}^{\mathtt{bool}}$ the ill-typed binding $\mathtt{F}^{\alpha\to\mathtt{bool}} \mapsto (\mathtt{plus}^{\mathtt{nat}\to\mathtt{nat}\to\mathtt{nat}}\,\mathtt{Z}^{\mathtt{nat}})^{\mathtt{nat}\to\mathtt{nat}}$ would be prevented by a failure of type unification. Another possibility is to embed *ad hoc* mechanisms at a lower implementation level to take care of type unification problems at run time in an efficient way, as proposed in [31] for implementing polymorphic typing in the HO logic language λ-Prolog [40]. In the particular case of *monomorphically* typed programs, the problem is easier to solve, because no type unification is needed. In this vein, a recent paper [1] proposes a Warren-like [49] type-preserving translation of HO programs into FO programs in a monomorphic FLP setting. This technique is useful to prune the search space and to ensure well-typed computations without dynamic type checking, but it fails to handle goals with polymorphic function-typed variables.

Full type annotations have the disadvantage that goal solving becomes more complex and less efficient, also for those computations that do not need any dynamic type checking. Therefore, some optimization techniques were proposed in [23, 22, 24] to alleviate this difficulty. Our aim here is to adopt a less costly solution to extend CLNC with dynamic type checking, avoiding type-annotations. We use goals G similar to those from [19], but extended with a type environment T, and the notion of solution will be also extended so that solutions can provide bindings both for the data variables occurring in G and for the type variables occurring in T. Those CLNC transformations that must compute a binding t for some HO variable F will perform dynamic type checking, using type unification to ensure that the type of t is compatible with $T(F)$. The CLNC transformations not related to HO variable bindings will perform no dynamic type checking. In case of a computation that involves no functional application of HO logic variables, the only overhead w.r.t. untyped CLNC consists in maintaining the type environment T of the current goals, which of course evolves along the computation. This is not too costly, because all the type assumptions

needed for T can be taken from the type environment of the initial goal and the type environments of the rewrite rules in the program, which are known in advance.

In the rest of this section we develop these ideas and we investigate the soundness, completeness and type preservation properties of the resulting lazy narrowing calculus. Since the eventual occurrence of opaque decomposition during a CLNC computation is undecidable, our soundness result guarantees a well-typed computed answer only under the assumption that no opaque decomposition steps have occurred in the computation. On the other hand, our completeness result only ensures the well-typed computation of those solutions whose correctness is witnessed by some type-annotated GORC proof.

4.1 Admissible Goals and Solutions

The lazy narrowing calculus CLNC from [19] works with goals that include both joinability statements $a \bowtie b$ to be solved and approximation statements $e \rightarrow t$ to represent delayed unifications, introduced by lazy narrowing steps. Moreover, goals in G include a *solved part* to represent the answer data substitution computed so far, and they satisfy a number of *goal invariants*. Here we must extend this class of goals, adding a *type environment* and a second solved part to represent an answer type substitution. Those (data and type) variables of a goal that have been introduced locally by previous CLNC steps will be viewed as *existential*. For technical convenience, in the rest of the paper we assume a countable set of existential variables, and we use the notation $(l \rightarrow r \Leftarrow C \;\square\; T) \in_{var} \mathcal{P}$ to indicate that $(l \rightarrow r \Leftarrow C \;\square\; T)$ is a renaming of some defining rule from $\mathcal{P}$, using fresh existential type variables in T and fresh existential data variables in the rewrite rule. We also write $a \asymp b$ for any one of the conditions $a \bowtie b$ or $b \bowtie a$. The formal definition of goal follows. More motivation and explanations can be found in [18, 47] for the untyped FO case.

Definition 10 (Admissible Goals)
Admissible goals have the form $G = P \;\square\; C \;\square\; S_d \;\square\; S_t \;\square\; T$, *where:*

1. *The* delayed part $P = e_1 \rightarrow t_1, \ldots, e_k \rightarrow t_k$ *is a multiset of approximation statements, with* $e_i \in Exp$, $t_i \in Pat$. *The set of* produced data variables *of* G *is defined as* $pvar(P) = var(t_1) \cup \cdots \cup var(t_k)$, *and the*

production relation *is defined over* $var(G)$ *by the condition* $X \gg_P Y$ *iff there is some* $1 \leq i \leq k$ *such that* $X \in var(e_i)$ *and* $Y \in var(t_i)$.

2. *The* unsolved part $C = a_1 \bowtie b_1, \ldots, a_l \bowtie b_l$ *is a multiset of joinability statements. The set of* demanded data variables *of* G *is defined as* $ddvar(C) = \{X \in DVar \mid X\ \overline{e}_m \asymp b \in C, \text{ for some } \overline{e}_m, b\}$.

3. *The* data solved part $S_d = \{X_1 \approx s_1, \ldots, X_n \approx s_n\}$ *is a set of equations in solved form, such that each* $s_i \in Pat$ *and each* X_i *occurs exactly once in* $P \ \square\ C \ \square\ S_d$. *We write* σ_d *for* $mgu(S_d)$.

4. *The* type solved part $S_t = \{\alpha_1 \approx \tau_1, \ldots, \alpha_m \approx \tau_m\}$ *is a set of type equations in solved form, such that each* $\tau_i \in Type$ *and each* α_i *occurs exactly once in* G. *We write* σ_t *for* $mgu(S_t)$.

5. $T = \{X_1 :: \tau_1, \ldots, X_p :: \tau_p\}$ *is called the* type environment *of* G. *We assume that* $var(P \ \square\ C \ \square\ S_d) \subseteq \{X_1, \ldots, X_p\} = dom(T)$.

6. G *must fulfill the following conditions, called* goal invariants*:*

   LN *The tuple* $(t_1, \ldots, t_k)$ *is linear.*

   EX *All produced data variables are existential.*

   NC *The transitive closure of the production relation* $\gg_P$ *is irreflexive.*

   SL *No produced variable enters the solved part:* $var(S_d) \cap pvar(P) = \emptyset$.

Note that any admissible goal verifies $(P \ \square\ C)\sigma_d = (P \ \square\ C)$ and $T\sigma_t = T$, because of the requirements in items 3 and 4 above. By convention, *initial goals* are of the form $G_0 = \emptyset \ \square\ C \ \square\ \emptyset \ \square\ \emptyset \ \square\ T_0$, and include no existential variables.

For typing purposes, goals can be viewed as "expressions" built from variables, symbols in the current signature and binary "operations" $(\bowtie)$, $(\rightarrow)$, $(\approx)$ of type $\alpha \rightarrow \alpha \rightarrow \texttt{bool}$ and $(,)$ of type $\texttt{bool} \rightarrow \texttt{bool} \rightarrow \texttt{bool}$, used in infix notation. We say that a goal $G = P \ \square\ C \ \square\ S_d \ \square\ S_t \ \square\ T$ is well-typed iff there is some $T \leq T'$ such that $T' \vdash_{WT} (P, C, S_d) :: \texttt{bool}$. In addition, G is called *type-closed* iff T itself can be taken as T'. More formally:

Definition 11 (Well-typed Goals)
Consider an admissible goal $G = P \ \square\ C \ \square\ S_d \ \square\ S_t \ \square\ T$ *and assume*

$TR(T, (P, C, S_d)) = (-, E)$, where TR is the type reconstruction algorithm from Subsection 2.2.

1. *G is called* well-typed *iff $TSol(E) \neq \emptyset$. If this is the case, we write $\hat{\sigma}_t = mgu(S_t,\ E)$, $\hat{T} = T\hat{\sigma}_t$ and $\hat{G} = P \ \square\ C \ \square\ S_d \ \square\ \hat{S}_t \ \square\ \hat{T}$.*

2. *A well-typed goal G is called* type-closed *iff $T = \hat{T}$. In the case that G is well-typed, but not type-closed, $\hat{G}$ is called the* type closure *of G.*

Note that a well-typed goal is not always type-closed, as shown next.

Example 7 (Type-closure of a Well-typed Goal)

1. A well-typed goal which is not type-closed:
   $G = \emptyset \ \square\ \mathtt{F\ X} \bowtie \mathtt{true} \ \square\ \emptyset \ \square\ \emptyset \ \square\ \{\mathtt{X} :: \alpha_1,\ \mathtt{F} :: \alpha_1 \to \alpha\}$.

2. Its type-closure:
   $\hat{G} = \emptyset \ \square\ \mathtt{F\ X} \bowtie \mathtt{true} \ \square\ \emptyset \ \square\ \alpha \approx \mathtt{bool} \ \square\ \{\mathtt{X} :: \alpha_1,\ \mathtt{F} :: \alpha_1 \to \mathtt{bool}\}$.

The following technical lemma gives a useful characterization of well-typed goals. A proof is included in the Appendix.

Lemma 13 (Characterization of Well-typed Goals)
For any admissible goal $G = P \ \square\ C \ \square\ S_d \ \square\ S_t \ \square\ T$, the three following conditions are equivalent:

(a) G is well-typed.

(b) There is some $T \leq T'$ such that $T' \vdash_{WT} (P, C, S_d) :: \mathtt{bool}$.

(c) There is some $T \leq T'$ such that

   (c1) For every $(e \to t) \in P$ there is some type τ such that $T' \vdash_{WT} e :: \tau$, $T' \vdash_{WT} t :: \tau$.

   (c2) For every $(a \bowtie b) \in C$ there is some type τ such that $T' \vdash_{WT} a :: \tau$, $T' \vdash_{WT} b :: \tau$.

   (c3) For every $(X \approx t) \in S_d$, $T' \vdash_{WT} t :: \tau$, where $\tau = T'(X)$.

   Moreover, T' can be always chosen as $\hat{T}$, if it exists. □

In the sequel, the notation $T \vdash_{WT} G$ will abbreviate $T \vdash_{WT} (P, C, S_d) :: \texttt{bool}$. According to Lemma 13, this indicates that G is well-typed and type-closed. Similarly, the abbreviation $\hat{T} \vdash_{WT} G$ can be used to indicate that G is a well-typed goal.

Next, we define solutions of admissible goals. According to our definition, solutions are proved to be correct by means of certain GORC proofs, called *witnesses*.

Definition 12 (Solutions and Well-typed Solutions)
Let $G = P \;\square\; C \;\square\; S_d \;\square\; S_t \;\square\; T$ be an admissible goal for a program $\mathcal{P}$.

1. *A* solution *of G is any data substitution $\theta_d \in DSub_\perp$ satisfying the following conditions:*
   (a) *$\theta_d(X)$ is a total pattern, for all $X \in dom(\theta_d) \setminus pvar(P)$.*
   (b) *$\theta_d \in Sol(S_d)$.*
   (c) *$\mathcal{P} \vdash_{GORC} (P \;\square\; C)\theta_d$, intended to mean that $\mathcal{P} \vdash_{GORC} \varphi\theta_d$ must hold for all $\varphi \in P \cup C$.*
2. *We write $Sol(G)$ for the set of all the solutions of G. Any multiset $\mathcal{M}$ containing one GORC proof for each statement $\varphi\theta_d$, $\varphi \in P \cup C$, is called a* witness *for $\theta_d \in Sol(G)$.*
3. *A* well-typed solution *of G is any pair (R, θ) formed by a type environment R and a substitution $\theta = (\theta_t, \theta_d)$ such that*
   (a) *$\theta_d \in Sol(G)$.*
   (b) *$dom(R) \subseteq ran(\theta_d) \setminus dom(T)$.*
   (c) *$\theta_t \in TSol(S_t)$.*
   (d) *$(T\theta_t \cup R) \vdash_{WT} T\theta$, intended to mean that $(T\theta_t \cup R) \vdash_{WT} X\theta_d :: \tau\theta_t$ must hold for all $(X :: \tau) \in T$.*
4. *We write $WTSol(G)$ for the set of all the well-typed solutions of G. A* witness *for (R, θ) is defined simply as a witness for $\theta \in Sol(G)$.*

Items 1 and 2 in the previous definition correspond essentially to our former notion of solution in the untyped setting from [19]. The rôle of the type environment R in well-typed solutions is to provide type assumptions for the new data variables introduced in $ran(\theta_d)$; see item 3 in the definition.

Of course, ill-typed goals can have solutions, in the sense of items 1 and 2. More strangely perhaps, some ill-typed goals have well-typed solutions. For example $R = \emptyset$, $\theta_t = \emptyset$ and $\theta_d = \emptyset$ give a well-typed solution for the ill-typed goal $G = \emptyset \ \square \ \texttt{tail [z]} \bowtie \texttt{tail [true]} \ \square \ \emptyset \ \square \ \emptyset \ \square \ \emptyset$.

In the case of well-typed solutions of well-typed goals, one expects the goal to remain well-typed when the solution is applied to it. As shown by the next lemma (proved in the Appendix) this can be guaranteed only for type-closed goals.

Lemma 14 (Well-typed Solutions and Type-closure)

1. *For any well-typed goal G, $WTSol(\hat{G}) \subseteq WTSol(G)$. The opposite inclusion is false in general.*

2. *Assume a well-typed and type-closed goal G and $(R, \theta) \in WTSol(G)$. Then $(T\theta_t \cup R) \vdash_{WT} G\theta_d$. In particular, for all $(a \bowtie b) \in C$ there is some $\tau' \in Type$ such that $(T\theta_t \cup R) \vdash_{WT} a\theta_d :: \tau'$ and $(T\theta_t \cup R) \vdash_{WT} b\theta_d :: \tau'$ (and analogously for $(e \to t) \in P$ and $(X \approx t) \in S_d$). These claims can fail if G is not type-closed.* □

In view of the previous lemma, we restrict our attention to well-typed solutions of well-typed and type-closed goals. We are also interested in *type-annotated witnesses* for such solutions, in the sense of the following definition:

Definition 13 (Type-annotated Witness)
Assume a well-typed and type-closed goal $G = P \ \square \ C \ \square \ S_d \ \square \ S_t \ \square \ T$ for a well-typed program $\mathcal{P}$. A type-annotated witness *of $(R, \theta) \in WTSol(G)$, if it exists, is obtained by considering the principal type annotation $(P \ \square \ C)^{\texttt{bool}}$ with implicit type environment T, and taking a multiset $\mathcal{M}$ of type-annotated GORC proofs for the type-annotated statement in $(P \ \square \ C)^{\texttt{bool}}\theta$. Note that the type-annotated "expression" $(P \ \square \ C)^{\texttt{bool}}\theta$ makes sense because of Lemma 5, and has implicit type environment $T\theta_t \cup R$.*

In the sequel, we write $TASol(G)$ for the set of all $(R, \theta) \in WTSol(G)$ which have a type-annotated witness. Trivially, $TASol(G) \subset WTSol(G)$. However, the inclusion is strict, as shown by the next example.

Example 8 (Type-annotated Witness)
Consider:

$$G = \emptyset \;\square\; \texttt{snd (tail [X])} \bowtie \texttt{snd (tail [Y])} \;\square\; \emptyset \;\square\; \emptyset \;\square\; \{\texttt{X} :: \alpha_1,\ \texttt{Y} :: \alpha_2\},$$

$R = \emptyset$, $\theta_t = \{\alpha_1 \mapsto \texttt{nat}, \alpha_2 \mapsto \texttt{bool}\}$ and $\theta_d = \{\texttt{X} \mapsto \texttt{z}, \texttt{Y} \mapsto \texttt{true}\}$. Note that G is well-typed and type-closed, and $(R, \theta) \in \mathit{WTSol}(G)$. However, no type-annotated witness exists due to Example 6(b).

As we will see in the next subsection, maintaining type-closed goals during CLNC computations would be as expensive as working with fully type-annotated goals. Therefore, CLNC is designed to avoid the computation of type closures whenever possible.

4.2 Lazy Narrowing Calculus

Now we are ready to extend the lazy narrowing calculus CLNC from [19] with dynamic type checking. We keep CLNC as the name of the new calculus. As in [19], the notation $G \vdash_{CLNC} G'$ means that G is transformed into G' in one step. The aim when using CLNC is to transform a well-typed, type-closed *initial goal* $G_0 = \emptyset \;\square\; C \;\square\; \emptyset \;\square\; \emptyset \;\square\; T_0$ into a well-typed *solved goal* $G_n = \emptyset \;\square\; \emptyset \;\square\; S_d \;\square\; S_t \;\square\; T_n$ with type closure $\hat{G}_n = \emptyset \;\square\; \emptyset \;\square\; S_d \;\square\; \hat{S}_t \;\square\; \hat{T}_n$, and to return $(T_n\hat{\sigma}_t \setminus T_0\hat{\sigma}_t, (\hat{\sigma}_t, \sigma_d))$ as the answer computed for G_0. A sequence of transformation steps $G_0 \vdash_{CLNC} \cdots \vdash_{CLNC} G_n$ going from an initial goal G_0 to a solved goal G_n, is called CLNC *derivation* and noted as $G_0 \vdash^*_{CLNC} G_n$. Due to Lemma 14, it is relevant to assume type-closedness for the initial goal G_0 and to compute the type closure of the final goal G_n before extracting the computed answer. Some intermediate goals G_i may be not type-closed, but nevertheless they stand for their closures $\hat{G}_i$ for the purpose of considering potential solutions. The need to include the type environments $(T_n\hat{\sigma}_t \setminus T_0\hat{\sigma}_t)$ as part of computed answers will be justified in Subsection 4.3.

Due to the convention that P and C are understood as multisets, CLNC assumes no particular *selection strategy* for choosing the goal statement to be processed in the next step. For writing failure rules we use **FAIL**, representing an irreducible inconsistent goal. We also use some hopefully self-explanatory abbreviations for tuples. In particular, $\overline{a}_m \bowtie \overline{b}_m$ stands for m new joinability statements $a_i \bowtie b_i$, and similarly for approximation statements. According to the notations introduced in Definition 10, CLNC rules

rely on the convention that S'_t is the set of type equations in solved form representing the unifier σ'_t. Moreover, some CLNC rules use the notation "$[\cdots]$" meaning an optional part of a goal, present only under certain conditions. Some other rules refer to the set $svar(e)$ of those data variables that occur in e at some position outside the scope of evaluable function calls. Formally, $svar(X) = \{X\}$ for any data variable X; for a rigid and passive expression $e = h\,\overline{e}_m$, $svar(e) = \bigcup_{i=1}^{m} svar(e_i)$; and $svar(e) = \emptyset$ in any other case.

In spite of their complex appearance, CLNC transformation rules are natural in the sense that they are designed to guess the shape of GORC derivations step by step. For instance, the transformations **NR1** and **GN** guess the shape of **OR** steps in GORC proofs. As a lazy narrowing calculus, CLNC emulates suspensions and sharing by means of the approximation statements in the delayed part of goals. This and other interesting features regarding occurs check and safe cases for eager variable elimination, have been discussed briefly in [19] for the untyped HO case, and more widely in [18, 47] for the untyped FO case. Here we focus on the treatment of dynamic type checking. Each CLNC transformation has attached certain side conditions, labelled with the symbols $\bullet$, $*$ and $\circ$, that must be checked before applying the transformation to the current goal G. In particular, those CLNC transformations whose name is marked with $\bigstar$ have a side condition of type $\circ$ that performs dynamic type checking. In all such cases, there is a candidate binding $X \mapsto t$ for some HO variable X. The type $T(X)$ assumed for X in the current goal's environment is compared to the principal type of t by means of type unification. In case of success, the type solved part of the goal is properly actualized, and the CLNC step can be performed. In case of failure, the CLNC step is forbidden for the particular binding $X \mapsto t$. Of course, this does not exclude the possibility to try a different application of the same CLNC transformation, with a different binding.

The CLNC Calculus

Rules for the Unsolved Part

Identity & Decomposition (ID) $\bigstar$ $(p \geq 0)$

$$P \;\square\; X\,\overline{a}_p \asymp X\,\overline{b}_p,\; C \;\square\; S_d \;\square\; S_t \;\square\; T$$
$$\vdash\quad (P \;\square\; \overline{a}_p \asymp \overline{b}_p,\; C \;\square\; S_d)[\rho_d,\; X \approx h\,\overline{V}_m] \;\square\; S'_t \;\square\; ([\overline{V}_m :: \overline{\tau}_m],\; \hat{T})\sigma'_t$$

- $X \notin pvar(P)$.
- $\circ$ $p = 0$ and $S'_t = S_t$; or [$p > 0$, $m \geq 0$, $h\,\overline{V}_m\,\overline{a}_p$ rigid and passive; $\overline{V}_m$ fresh existential variables; $\sigma'_t = mgu(\hat{S}_t,\ \tau \approx \tau')$ where $\tau' = \hat{T}(X)$, $h :: \overline{\tau}_m \to \tau \in_{var} \Sigma$.
- $*$ $\rho_d = \{X \mapsto h\,\overline{V}_m\}$.]

Decomposition (DC1)

$$P \sqcap h\,\overline{a}_m \asymp h\,\overline{b}_m,\ C \sqcap S_d \sqcap S_t \sqcap T$$
$$\vdash\quad P \sqcap \overline{a}_m \asymp \overline{b}_m,\ C \sqcap S_d \sqcap S_t \sqcap T$$

- $h\,\overline{a}_m$, $h\,\overline{b}_m$ rigid and passive.

Binding & Decomposition (BD) ★ $(k \geq 0)$

$$P \sqcap X\,\overline{a}_k \asymp s\,\overline{b}_k,\ C \sqcap S_d \sqcap S_t \sqcap T$$
$$\vdash\quad (P \sqcap \overline{a}_k \asymp \overline{b}_k,\ C \sqcap S_d)\rho_d,\ X \approx s \sqcap S'_t \sqcap \hat{T}\sigma'_t$$

- $s \in Pat$; $X \notin var(s)$; $X \notin pvar(P)$; $var(s) \cap pvar(P) = \emptyset$.
- $*$ $\rho_d = \{X \mapsto s\}$.
- $\circ$ $k = 0$ and $S'_t = S_t$; or $k > 0$, $s\overline{b}_k$ rigid and passive, and $\sigma'_t = mgu(\hat{S}_t, \tau \approx \tau')$ where $\tau = \hat{T}(X)$, τ' type-annotation of s in $PA(\hat{T}, s\,\overline{b}_k)$.

Imitation & Decomposition (IM) ★ $(k \geq 0)$

$$P \sqcap X\,\overline{a}_k \asymp h\,\overline{e}_m\,\overline{b}_k,\ C \sqcap S_d \sqcap S_t \sqcap T$$
$$\vdash\quad (P \sqcap \overline{V}_m\,\overline{a}_k \asymp \overline{e}_m\,\overline{b}_k,\ C \sqcap S_d)\rho_d,\ X \approx h\,\overline{V}_m \sqcap S'_t \sqcap (\overline{V}_m :: \overline{\tau}_m,\ \hat{T})\sigma'_t$$

- $h\,\overline{e}_m\,\overline{b}_k$ rigid and passive; $X \notin pvar(P)$; $X \notin svar(h\,\overline{e}_m)$; $\overline{V}_m$ fresh existential variables; **BD** not applicable.
- $*$ $\rho_d = \{X \mapsto h\,\overline{V}_m\}$.
- $\circ$ $k = 0$ and $S'_t = S_t$; or $k > 0$ and $\sigma'_t = mgu(\hat{S}_t, \tau \approx \tau')$ where $\tau' = \hat{T}(X)$, $h :: \overline{\tau}_m \to \tau \in_{var} \Sigma$.

Outer Narrowing (NR1) $(k \geq 0)$

$$P \;\square\; f\,\overline{e}_n\,\overline{a}_k \asymp b,\; C \;\square\; S_d \;\square\; S_t \;\square\; T$$
$$\vdash\;\; \overline{e}_n \to \overline{t}_n,\; P \;\square\; C',\; r\,\overline{a}_k \asymp b,\; C \;\square\; S_d \;\square\; S_t \;\square\; T',\; T$$

- $(f\,\overline{t}_n \to r \Leftarrow C' \;\square\; T') \in_{var} \mathcal{P}$.

Guess & Outer Narrowing (GN) ★ $(k \geq 0)$

$$P \;\square\; X\,\overline{e}_q\,\overline{a}_k \asymp b,\; C \;\square\; S_d \;\square\; S_t \;\square\; T$$
$$\vdash\;\; (\overline{e}_q \to \overline{s}_q,\; P \;\square\; C',\; r\,\overline{a}_k \asymp b,\; C \;\square\; S_d)\rho_d,\; X \approx f\,\overline{t}_p \;\square\; S'_t \;\square\; (T',\; \hat{T})\sigma'_t$$

- $q > 0$; $X \notin pvar(P)$; $(f\,\overline{t}_p\,\overline{s}_q \to r \Leftarrow C' \;\square\; T') \in_{var} \mathcal{P}$.
  - $\rho_d = \{X \mapsto f\,\overline{t}_p\}$.
  - $\sigma'_t = mgu(\hat{S}_t,\; \tau' \approx \overline{\lambda}_q \to \tau)$ where $(f :: \overline{\tau}_p \to \overline{\lambda}_q \to \tau) \in_{var} \Sigma$, $\tau' = \hat{T}(X)$.

Guess & Decomposition (GD) ★

$$P \;\square\; X\,\overline{a}_p \asymp Y\,\overline{b}_q, C \;\square\; S_d \;\square\; S_t \;\square\; T$$
$$\vdash\;\; (P \;\square\; \overline{V}_{m-p}, \overline{a}_p \asymp \overline{W}_{m-q}\,\overline{b}_q, C \;\square\; S_d)\rho_d,\; X \approx h\,\overline{V}_{m-p},\; Y \approx h\,\overline{W}_{m-q} \;\square\; S'_t \;\square\; (\overline{V}_{m-p} :: \overline{\tau'}_{m-p},\; \overline{W}_{m-q} :: \overline{\tau''}_{m-q},\; \hat{T})\sigma'_t$$

- $p{+}q > 0$; X, Y different; $X, Y \notin pvar(P)$; $\overline{V}_{m-p}, \overline{W}_{m-q}$ fresh existential variables; $(h\,\overline{V}_{m-p}\,\overline{a}_p)$, $(h\,\overline{W}_{m-q}\,\overline{b}_q)$ rigid and passive.
  - $\rho_d = \{X \mapsto h\,\overline{V}_{m-p}, Y \mapsto h\,\overline{W}_{m-q}\}$.
  - $\sigma'_t = mgu(\hat{S}_t,\; \tau' \approx \overline{\lambda'}_p \to \tau'_0,\; \tau'' \approx \overline{\lambda''}_q \to \tau''_0)$ where $\tau' = \hat{T}(X)$, $\tau'' = \hat{T}(Y)$, $(h :: \overline{\tau'}_{m-p} \to \overline{\lambda'}_p \to \tau'_0), (h :: \overline{\tau''}_{m-q} \to \overline{\lambda''}_q \to \tau''_0) \in_{var} \Sigma$, two fresh variants of the principal type of h, sharing no type variables.

Conflict (CF1)

$$P \;\square\; h\,\overline{a}_p \asymp h'\,\overline{b}_q,\; C \;\square\; S_d \;\square\; S_t \;\square\; T \;\;\vdash\;\; \textbf{FAIL}$$

- $h \neq h'$ or $p \neq q$; $h\,\overline{a}_p$, $h'\,\overline{b}_q$ rigid and passive.

Cycle (CY)

$$P \;\square\; X \asymp a,\; C \;\square\; S_d \;\square\; S_t \;\square\; T \;\; \vdash \;\; \textbf{FAIL}$$

- $X \neq a$ and $X \in svar(a)$.

Rules for the Delayed Part

Decomposition (DC2)

$$h\,\overline{e}_m \to h\,\overline{t}_m,\; P \;\square\; C \;\square\; S_d \;\square\; S_t \;\square\; T$$
$$\vdash \;\; \overline{e}_m \to \overline{t}_m,\; P \;\square\; C \;\square\; S_d \;\square\; S_t \;\square\; T$$

- $h\,\overline{e}_m$ rigid and passive.

Output Binding & Decomposition (OB) ★ $(k \geq 0)$

$$X\,\overline{e}_k \to h\,\overline{t}_m\,\overline{s}_k,\; P \;\square\; C \;\square\; S_d \;\square\; S_t \;\square\; T$$
$$\vdash \;\; (\overline{e}_k \to \overline{s}_k,\; P \;\square\; C \;\square\; S_d)\rho_d,\; [X \approx h\,\overline{t}_m] \;\square\; S'_t \;\square\; \hat{T}\sigma'_t$$

- $[X \notin pvar(P).]$
- $*$ $\rho_d = \{X \mapsto h\,\overline{t}_m\}$.
- $\circ$ $k = 0$ and $S'_t = S_t$; or $k > 0$ and $\sigma'_t = mgu(\hat{S}_t,\, \tau \approx \tau')$ where $\tau' = \hat{T}(X)$, $(h :: \overline{\tau}_m \to \tau) \in_{var} \Sigma$.

Input Binding (IB)

$$t \to X,\; P \;\square\; C \;\square\; S_d \;\square\; S_t \;\square\; T$$
$$\vdash \;\; (P \;\square\; C)\rho_d \;\square\; S_d \;\square\; S_t \;\square\; T$$

- $t \in Pat$.
- $*$ $\rho_d = \{X \mapsto t\}$.

Input Imitation (IIM)

$$h\,\overline{e}_m \to X,\; P \;\square\; C \;\square\; S_d \;\square\; S_t \;\square\; T$$
$$\vdash \;\; (\overline{e}_m \to \overline{V}_m,\; P \;\square\; C)\rho_d \;\square\; S_d \;\square\; S_t \;\square\; (\overline{V}_m :: \overline{\tau}_m),\; T$$

- $h\,\overline{e}_m \notin Pat$ rigid and passive; $X \in ddvar(C)$; $\overline{V}_m$ fresh existential variables.

$*$ $\rho_d = \{X \mapsto h\,\overline{V}_m\}$.

$\circ$ $(h :: \overline{\tau}_m \to \tau) \in_{var} \Sigma$.

Elimination (EL)

$$\begin{array}{l} e \to X,\ P \,\square\, C \,\square\, S_d \,\square\, S_t \,\square\, T \\ \vdash\ \ P \,\square\, C \,\square\, S_d \,\square\, S_t \,\square\, T \end{array}$$

- $X \notin var(P \,\square\, C)$.

Outer Narrowing (NR2) $(k \geq 0)$

$$\begin{array}{l} f\,\overline{e}_n\,\overline{a}_k \to t,\ P \,\square\, C \,\square\, S_d \,\square\, S_t \,\square\, T \\ \vdash\ \ \overline{e}_n \to \overline{t}_n,\ r\,\overline{a}_k \to t,\ P \,\square\, C',\ C \,\square\, S_d \,\square\, S_t \,\square\, T',\ T \end{array}$$

- $(t \notin DVar$ or $t \in ddvar(C))$; $(f\,\overline{t}_n \to r \Leftarrow C' \,\square\, T') \in_{var} \mathcal{P}$.

Output Guess & Outer Narrowing (OGN) ★ $(k \geq 0)$

$$\begin{array}{l} X\,\overline{e}_q\,\overline{a}_k \to t,\ P \,\square\, C \,\square\, S_d \,\square\, S_t \,\square\, T \\ \vdash\ \ (\overline{e}_q \to \overline{s}_q,\ r\,\overline{a}_k \to t,\ P \,\square\, C',\ C \,\square\, S_d)\rho_d,\ [X \approx f\,\overline{t}_p] \,\square\, S'_t \,\square \\ \qquad (T',\ \hat{T})\sigma'_t \end{array}$$

- $q > 0$; $[X \notin pvar(P)]$; $(f\,\overline{t}_p\,\overline{s}_q \to r \Leftarrow C' \,\square\, T') \in_{var} \mathcal{P}$; $t \notin DVar$ or $t \in ddvar(C)$.

$*$ $\rho_d = \{X \mapsto f\,\overline{t}_p\}$.

$\circ$ $\sigma'_t = mgu(\hat{S}_t,\ \tau' \approx \overline{\lambda}_q \to \tau)$ where $(f :: \overline{\tau}_p \to \overline{\lambda}_q \to \tau) \in_{var} \Sigma$, $\tau' = \hat{T}(X)$.

Output Guess & Decomposition (OGD) ★

$$\begin{array}{l} X\,\overline{e}_q \to Y,\ P \,\square\, C \,\square\, S_d \,\square\, S_t \,\square\, T \\ \vdash\ \ (\overline{e}_q \to \overline{W}_q,\ P \,\square\, C \,\square\, S_d)\rho_d,\ [X \approx h\,\overline{V}_p] \,\square\, S'_t \,\square \\ \qquad (\overline{V}_p :: \overline{\tau}_p,\ \overline{W}_q :: \overline{\lambda}_q,\ \hat{T})\sigma'_t \end{array}$$

- $q > 0$; $Y \in ddvar(C)$; $[X \notin pvar(P)]$; $\overline{V}_p$, $\overline{W}_q$ fresh existential variables; $(h\,\overline{V}_p\,\overline{W}_q)$ rigid and passive.

$*$ $\rho_d = \{X \mapsto h\,\overline{V}_p, Y \mapsto (h\,\overline{V}_p\,\overline{W}_q)\}$.

$\circ$ $\sigma'_t = mgu(\hat{S}_t,\ \tau' \approx \overline{\lambda}_q \to \tau,\ \tau'' \approx \tau)$ where $(h :: \overline{\tau}_p \to \overline{\lambda}_q \to \tau) \in_{var} \Sigma$, $\tau' = \hat{T}(X)$, $\tau'' = \hat{T}(Y)$.

Conflict (CF2)

$$h\,\overline{a}_p \to h'\,\overline{t}_q,\ P \sqcap C \sqcap S_d \sqcap S_t \sqcap T \ \vdash\ \textbf{FAIL}$$

- $h \neq h'$ or $p \neq q$; $h\,\overline{a}_p$ rigid and passive.

There is a simple relationship between the current presentation of CLNC and the untyped version from [19]. Dropping $S_t \sqcap T$ from our current goals, and omitting all the side conditions which refer to types in our current CLNC transformations, gives back essentially the untyped goals and the CLNC transformations from [19], except for some minor modifications which we found convenient to introduce while working out the present soundness and completeness results. Due to this fact, the soundness and completeness results given for untyped CLNC in [19] are still valid in our present setting, with respect to "forgetful" CLNC derivations which omit dynamic checking. Of course, we are presently not interested in such untyped derivations, and the current CLNC transformations are intended for well-typed goals. Moreover, the CLNC transformations marked with ★ must be applied to the type closure $\hat{G}$ of the current goal G, rather than to G itself, in the case that G is not type-closed. For this reason, the formulation of the ★ transformations uses $\hat{S}_t$ and $\hat{T}$, rather than S_t and T, for building the transformed goal. This caution is very important, because several CLNC transformations do not preserve type-closedness of goals, and the type environment in a type-closed goal bears more precise information in general. Therefore, for goals $G \neq \hat{G}$, the direct application of ★ to G may leave some type errors undetected. In the case of CLNC derivations involving no functional applications of HO logic variables, ★ transformations are never applied, and the costly computation of type closures can be avoided.

Note that even the ★ transformations do not perform dynamic type checking in some cases, as indicated in their formulation. For instance, **ID** in the case $p = 0$ simply eliminates an identity $X \bowtie X$ from the current goal. Similar comments apply to **BD**, **IM** and **OB** in the case $k = 0$.

At this point, we can prove that the detection of eventual opaque decomposition in CLNC computations is an undecidable problem. More precisely,

let us define the *Opaque Decomposition Problem* (ODP, for short) as follows: given a well-typed CRWL program $\mathcal{P}$ and a well-typed and type-closed initial goal G for $\mathcal{P}$, tell if solving G w.r.t. $\mathcal{P}$ by using CLNC can eventually lead to an opaque decomposition step. The next theorem, proved in the Appendix, shows that the restriction of ODP to a quite simple class of programs and goals is undecidable.

Theorem 4 (Undecidability of ODP)
Let us say that a CRWL program $\mathcal{P}$ is simple *iff all the defining rules in $\mathcal{P}$ are unconditional and have data terms as patterns in their left-hand sides. Analogously, let us consider simple initial goals of the form $\emptyset \;\square\; f\,\overline{t}_n \bowtie g\,\overline{s}_m \;\square\; \emptyset \;\square\; \emptyset \;\square\; \emptyset$, where $f \in FS^n$, $g \in FS^m$ and t_i, s_j are closed data terms. Then, the restriction of ODP to simple programs and simple initial goals is undecidable.* □

Note that Theorem 4 does not mean that CLNC is useless for the purpose of avoiding ill-typed computations. Type errors due to ill-typed bindings of HO logic variables are always avoided. Regarding type errors due to opaque decompositions, our feeling is that they occur rarely in practice. Obviously, the opaque decomposition problem cannot arise in FO CRWL programs, where transparent data terms are always used in place of possibly opaque patterns. In fact, type preservation results for lazy narrowing computations in such a setting are known [5, 4]. Beyond the FO case (which includes, of course, FO logic programming) opaque decompositions are possible, but sometimes they do not cause any type errors. We conjecture that the impossibility of run-time type errors due to opaque decomposition can be formally proved for a restricted subclass of HO CRWL programs and goals. More precisely, we have in mind the class of simple (but possible higher-order) programs used in the proof of Theorem 4, and goals of the form $\emptyset \;\square\; e \bowtie R \;\square\; \emptyset \;\square\; \emptyset \;\square\; \{R :: \tau\}$ where e is a closed expression of type τ. This essentially corresponds to pure functional programming.

To investigate the previous conjecture is beyond the scope of this paper. In any case, an actual implementation of CLNC *could* (and *should*) raise a warning to the user whenever some opaque decomposition step has occurred during the computation. This *may* (but not *must*) indicate a run time type error.

The soundness and completeness of CLNC are investigated in subsection 4.3. In the rest of this subsection we illustrate the behaviour of CLNC

by means of examples, all of them using well-typed functions defined in the program from Example 2. Some of the examples present a complete CLNC derivation. In such cases, the initial goal is always well-typed and type-closed, and the part of the current goal which is transformed at each step is underlined.

Our first example shows that type-closedness of goals is not invariant under CLNC transformations. It can be lost because of an opaque decomposition step, as in item (a), or also because of a narrowing step, as in item (b). In the derivation (b), the new goal is not type closed because the principal type of the defined function `head` (introduced by the narrowing step) has not been unified with the type expected by the goal's environment. In both cases, the final solved goal must be type-closed before extracting the computed answer, which would be not a well-typed solution otherwise.

Example 9 (Type-closedness of Goals is not CLNC Invariant)

(a) *Opaque Decomposition*

$$G_0 = \emptyset \;\square\; \underline{\mathtt{snd\,X} \bowtie \mathtt{snd\,Y}} \;\square\; \emptyset \;\square\; \emptyset \;\square\; \{\mathtt{X} :: \alpha,\, \mathtt{Y} :: \beta\} \quad \vdash_{\mathbf{DC1}}$$

$$G_1 = \emptyset \;\square\; \underline{\mathtt{X} \bowtie \mathtt{Y}} \;\square\; \emptyset \;\square\; \emptyset \;\square\; \{\mathtt{X} :: \alpha,\, \mathtt{Y} :: \beta\} \quad \vdash_{\mathbf{BD}}$$

$$G_2 = \emptyset \;\square\; \emptyset \;\square\; \mathtt{X} \approx \mathtt{Y} \;\square\; \emptyset \;\square\; \{\mathtt{X} :: \alpha,\, \mathtt{Y} :: \beta\} \quad \not\equiv$$

$$\hat{G}_2 = \emptyset \;\square\; \emptyset \;\square\; \mathtt{X} \approx \mathtt{Y} \;\square\; \alpha \approx \beta \;\square\; \{\mathtt{X}, \mathtt{Y} :: \beta\}$$

Computed answer, restricted to variables in G_0:

$$(\emptyset, (\{\alpha \mapsto \beta\}, \{\mathtt{X} \mapsto \mathtt{Y}\})).$$

(b) *Outer Narrowing*

$$G_0 = \emptyset \;\square\; \underline{\mathtt{head\,L} \bowtie \mathtt{H}} \;\square\; \emptyset \;\square\; \emptyset \;\square\; \{\mathtt{L} :: [\alpha],\, \mathtt{H} :: \alpha\} \quad \vdash_{\mathbf{NR1}}$$

$$G_1 = \underline{\mathtt{L} \to \mathtt{[X|Xs]}} \;\square\; \mathtt{X} \bowtie \mathtt{H} \;\square\; \emptyset \;\square\; \emptyset \;\square\; \{\mathtt{L} :: [\alpha],\, \mathtt{H} :: \alpha,\, \mathtt{Xs} :: [\beta],\, \mathtt{X} :: \beta\} \quad \vdash_{\mathbf{OB}}$$

$$G_2 = \emptyset \;\square\; \underline{\mathtt{X} \bowtie \mathtt{H}} \;\square\; \mathtt{L} \approx \mathtt{[X|Xs]} \;\square\; \emptyset \;\square\; \{\mathtt{L} :: [\alpha],\, \mathtt{H} :: \alpha,\, \mathtt{Xs} :: [\beta],\, \mathtt{X} :: \beta\} \quad \vdash_{\mathbf{BD}}$$

$$G_3 = \emptyset \;\square\; \emptyset \;\square\; \mathtt{L} \approx \mathtt{[H|Xs]},\, \mathtt{X} \approx \mathtt{H} \;\square\; \emptyset \;\square\; \{\mathtt{L} :: [\alpha],\, \mathtt{H} :: \alpha,\, \mathtt{Xs} :: [\beta],\, \mathtt{X} :: \beta\} \quad \not\equiv$$

$$\hat{G}_3 = \emptyset \;\square\; \emptyset \;\square\; \mathtt{L} \approx \mathtt{[H|Xs]},\, \mathtt{X} \approx \mathtt{H} \;\square\; \beta \approx \alpha \;\square\; \{\mathtt{L} :: [\alpha],\, \mathtt{H} :: \alpha,\, \mathtt{Xs} :: [\alpha],\, \mathtt{X} :: \alpha\}$$

Computed answer, restricted to variables in G_0:

$$(\{\texttt{Xs} :: [\alpha]\}, (\emptyset, \{\texttt{L} \mapsto \texttt{[H|Xs]}\})).$$

Our second example shows a complete CLNC derivation where no HO variables occur and no dynamic type checking is needed.

Example 10 (A Well-typed Computation without Dynamic Type Checking)

$G_0 = \emptyset \;\square\; \underline{\texttt{map (plus X) [Y]} \bowtie \texttt{[s z]}} \;\square\; \emptyset \;\square\; \emptyset \;\square\; \{\texttt{X}, \texttt{Y} :: \texttt{nat}\} \;\vdash_{\mathbf{NR1}}$

$\underline{\texttt{(plus X)} \to \texttt{F},\ \texttt{[Y|[]]} \to \texttt{[X1|X1s]}} \;\square\; \texttt{[F X1|map F X1s]} \bowtie \texttt{[s z]} \;\square\;$
$\emptyset \;\square\; \emptyset \;\square\; \{\texttt{X}, \texttt{Y} :: \texttt{nat};\ \texttt{X1s} :: [\alpha];\ \texttt{X1} :: \alpha;\ \texttt{F} :: \alpha \to \beta\} \;\vdash_{\mathbf{IB,DC2}}$

$\underline{\texttt{Y} \to \texttt{X1},\ \texttt{[]} \to \texttt{X1s}} \;\square\; \texttt{[(plus X) X1|map (plus X) X1s]} \bowtie \texttt{[s z]} \;\square\; \emptyset$
$\;\square\; \emptyset \;\square\; \{\texttt{X}, \texttt{Y} :: \texttt{nat};\ \texttt{X1s} :: [\alpha];\ \texttt{X1} :: \alpha;\ \texttt{F} :: \alpha \to \beta\} \;\vdash_{\mathbf{IB}^2}$

$\emptyset \;\square\; \underline{\texttt{[plus X Y|map (plus X) []]} \bowtie \texttt{[s z|[]]}} \;\square\; \emptyset \;\square\; \emptyset \;\square$
$\{\texttt{X}, \texttt{Y} :: \texttt{nat};\ \texttt{X1s} :: [\alpha];\ \texttt{X1} :: \alpha;\ \texttt{F} :: \alpha \to \beta\} \;\vdash_{\mathbf{DC1}}$

$\emptyset \;\square\; \underline{\texttt{plus X Y} \bowtie \texttt{s z}},\ \texttt{map (plus X) []} \bowtie \texttt{[]} \;\square\; \emptyset \;\square\; \emptyset \;\square$
$\{\texttt{X}, \texttt{Y} :: \texttt{nat};\ \texttt{X1s} :: [\alpha];\ \texttt{X1} :: \alpha;\ \texttt{F} :: \alpha \to \beta\} \;\vdash_{\mathbf{NR1}}$

$\underline{\texttt{X} \to \texttt{(s X2)},\ \texttt{Y} \to \texttt{Y2}} \;\square\; \texttt{s (plus X2 Y2)} \bowtie \texttt{s z},\ \texttt{map (plus X) []} \bowtie \texttt{[]}$
$\;\square\; \emptyset \;\square\; \emptyset \;\square\; \{\texttt{X}, \texttt{Y}, \texttt{X2}, \texttt{Y2} :: \texttt{nat};\ \texttt{X1s} :: [\alpha];\ \texttt{X1} :: \alpha;\ \texttt{F} :: \alpha \to \beta\}$
$\vdash_{\mathbf{OB,IB}}$

$\emptyset \;\square\; \underline{\texttt{s (plus X2 Y)} \bowtie \texttt{s z}},\ \texttt{map (plus (s X2))} \bowtie \texttt{[]} \;\square\; \texttt{X} \approx \texttt{s X2} \;\square$
$\emptyset \;\square\; \{\texttt{X}, \texttt{Y}, \texttt{X2}, \texttt{Y2} :: \texttt{nat};\ \texttt{X1s} :: [\alpha];\ \texttt{X1} :: \alpha;\ \texttt{F} :: \alpha \to \beta\} \;\vdash_{\mathbf{DC1}}$

$\emptyset \;\square\; \underline{\texttt{plus X2 Y} \bowtie \texttt{z}},\ \texttt{map (plus (s X2))} \bowtie \texttt{[]} \;\square\; \texttt{X} \approx \texttt{s X2} \;\square\; \emptyset \;\square$
$\{\texttt{X}, \texttt{Y}, \texttt{X2}, \texttt{Y2} :: \texttt{nat};\ \texttt{X1s} :: [\alpha];\ \texttt{X1} :: \alpha;\ \texttt{F} :: \alpha \to \beta\} \;\vdash_{\mathbf{NR1}}$

$\underline{\texttt{X2} \to \texttt{z},\ \texttt{Y} \to \texttt{Y3}} \;\square\; \texttt{Y3} \bowtie \texttt{z},\ \texttt{map (plus (s X2))} \bowtie \texttt{[]} \;\square\; \texttt{X} \approx \texttt{s X2}$
$\;\square\; \emptyset \;\square\; \{\texttt{X}, \texttt{Y}, \texttt{X2}, \texttt{Y2}, \texttt{Y3} :: \texttt{nat};\ \texttt{X1s} :: [\alpha];\ \texttt{X1} :: \alpha;\ \texttt{F} :: \alpha \to \beta\}$
$\vdash_{\mathbf{OB,IB}}$

$\emptyset \;\square\; \underline{\texttt{Y} \bowtie \texttt{z}},\texttt{map (plus (s z))} \bowtie \texttt{[]} \;\square\; \texttt{X} \approx \texttt{s z},\ \texttt{X2} \approx \texttt{z} \;\square\; \emptyset \;\square$
$\{\texttt{X}, \texttt{Y}, \texttt{X2}, \texttt{Y2}, \texttt{Y3} :: \texttt{nat};\ \texttt{X1s} :: [\alpha];\ \texttt{X1} :: \alpha;\ \texttt{F} :: \alpha \to \beta\} \;\vdash_{\mathbf{BD}}$

$\emptyset \;\square\; \underline{\texttt{map (plus (s z) []} \bowtie \texttt{[]}} \;\square\; \texttt{X} \approx \texttt{s z},\ \texttt{X2} \approx \texttt{z},\ \texttt{Y} \approx \texttt{z} \;\square\; \emptyset \;\square$
$\{\texttt{X}, \texttt{Y}, \texttt{X2}, \texttt{Y2}, \texttt{Y3} :: \texttt{nat};\ \texttt{X1s} :: [\alpha];\ \texttt{X1} :: \alpha;\ \texttt{F} :: \alpha \to \beta\} \;\vdash_{\mathbf{NR1}}$

$$\underline{\texttt{(plus (s z))} \to \texttt{F1},\ \texttt{[]} \to \texttt{[]} \ \square\ \texttt{[]} \bowtie \texttt{[]} \ \square\ \texttt{X} \approx \texttt{s z},\ \texttt{X2} \approx \texttt{z},\ \texttt{Y} \approx \texttt{z}}$$
$$\square\ \emptyset\ \square\ \{\texttt{X1s} :: [\alpha];\ \texttt{X}, \texttt{Y}, \texttt{X2}, \texttt{Y2}, \texttt{Y3} :: \texttt{nat};\ \texttt{X1} :: \alpha;\ \texttt{F} :: \alpha \to \beta;$$
$$\texttt{F1} :: \alpha_1 \to \beta_1\} \quad \vdash_{\mathbf{IB},\mathbf{DC2},\mathbf{DC1}}$$
$$\emptyset\ \square\ \emptyset\ \square\ \texttt{X} \approx \texttt{s z},\ \texttt{X2} \approx \texttt{z},\ \texttt{Y} \approx \texttt{z}\ \square\ \emptyset\ \square\ \{\texttt{X}, \texttt{Y}, \texttt{X2}, \texttt{Y2}, \texttt{Y3} :: \texttt{nat};$$
$$\texttt{X1s} :: [\alpha];\ \texttt{X1} :: \alpha;\ \texttt{F} :: \alpha \to \beta;\ \texttt{F1} :: \alpha_1 \to \beta_1\} = G_n = \hat{G}_n$$

Computed answer, restricted to variables in G_0:

$(\emptyset, (\emptyset, \{\texttt{X} \mapsto \texttt{s z}, \texttt{Y} \mapsto \texttt{z}\}))$.

Finally, we present a series of examples designed to show that the dynamic type checking side conditions embodied in CLNC are really needed. There is a different example for each of the eight transformations marked with ★, which can perform dynamic type checking. All the examples show that type errors can occur if dynamic type checking is omitted, or applied to a goal which is not type-closed. Some of the ★ marked CLNC transformations are such that a type error caused by opaque decompositions can escape from dynamic type checking. The examples do also illustrate these situations.

Example 11 (CLNC Transformation ID)

1. Type-closure and dynamic checking are necessary. Consider $G \neq \hat{G}$, given as:

$$G = \emptyset\ \square\ \texttt{X Y} \bowtie \texttt{X (not Z)}\ \square\ \emptyset\ \square\ \emptyset\ \square\ \{\texttt{X} :: \alpha \to \beta,\ \texttt{Y} :: \alpha,\ \texttt{Z} :: \gamma\}$$
$$\hat{G} = \emptyset\ \square\ \texttt{X Y} \bowtie \texttt{X (not Z)}\ \square\ \emptyset\ \square\ \alpha \approx \texttt{bool},\ \gamma \approx \texttt{bool}\ \square$$
$$\{\texttt{X} :: \texttt{bool} \to \beta,\ \texttt{Y} :: \texttt{bool},\ \texttt{Z} :: \texttt{bool}\}$$

   Dynamic type checking prevents the application of **ID** to $\hat{G}$ with binding $\{\texttt{X} \mapsto \texttt{twice}\}$, since $\hat{T}(\texttt{X}) = \texttt{bool} \to \beta$, $\texttt{twice} :: (\alpha' \to \alpha') \to \alpha' \to \alpha'$ and $mgu(\texttt{bool} \to \beta \approx (\alpha' \to \alpha') \to \alpha' \to \alpha') = \textbf{FAIL}$.

   In the case of G, dynamic type checking does not prevent a type error. One gets: $T(\texttt{X}) = \alpha \to \beta$, $\texttt{twice} :: (\alpha' \to \alpha') \to \alpha' \to \alpha'$ and $mgu(\alpha \to \beta \approx (\alpha' \to \alpha') \to \alpha' \to \alpha') = \{\alpha \mapsto (\alpha' \to \alpha'), \beta \mapsto (\alpha' \to \alpha')\}$ and hence $G \vdash_{\mathbf{ID},\text{wrong}} G'$, where

$$G' = \emptyset\ \square\ \texttt{Y} \bowtie \texttt{not Z}\ \square\ \texttt{X} \approx \texttt{twice}\ \square\ \alpha \approx \alpha' \to \alpha',\ \beta \approx \alpha' \to \alpha'$$
$$\square\ \{\texttt{X} :: (\alpha' \to \alpha') \to \alpha' \to \alpha',\ \texttt{Y} :: \alpha' \to \alpha',\ \texttt{Z} :: \gamma\}$$

Note that G' is ill-typed, although it admits an untyped solution: $\sigma_d = \{\mathtt{X} \mapsto \mathtt{twice}, \mathtt{Y} \mapsto \mathtt{false}, \mathtt{Z} \mapsto \mathtt{true}\}$.

2. Opaque decomposition escapes from dynamic type checking. This is not possible for transformation **ID**. Since $\mathtt{X}\ \overline{a_p} \asymp \mathtt{X}\ \overline{b_p}$ is well-typed w.r.t. $\hat{T}$, the type $\hat{T}(\mathtt{X})$ forces the types of $\overline{a_p}$ and $\overline{b_p}$ to be the same.

Example 12 (CLNC Transformation BD)

1. Type-closure and dynamic checking are necessary. Consider $G \neq \hat{G}$, given as:

$$G = \emptyset\ \square\ \mathtt{X\ Y} \bowtie \mathtt{s\ N},\ \mathtt{negate\ Y} \bowtie \mathtt{Z}\ \square\ \emptyset\ \square\ \emptyset\ \square \\ \{\mathtt{X} :: \alpha \to \mathtt{nat},\ \mathtt{Y} :: \mathtt{[bool]},\ \mathtt{Z} :: \mathtt{[bool]},\ \mathtt{N} :: \mathtt{nat}\}$$

$$\hat{G} = \emptyset\ \square\ \mathtt{X\ Y} \bowtie \mathtt{s\ N},\ \mathtt{negate\ Y} \bowtie \mathtt{Z}\ \square\ \emptyset\ \square\ \alpha \approx \mathtt{[bool]}\ \square \\ \{\mathtt{X} :: \mathtt{[bool]} \to \mathtt{nat},\ \mathtt{Y} :: \mathtt{[bool]},\ \mathtt{Z} :: \mathtt{[bool]},\ \mathtt{N} :: \mathtt{nat}\}$$

   Dynamic type checking prevents the application of **BD** to $\hat{G}$ with binding $\{\mathtt{X} \mapsto \mathtt{s}\}$, since $\hat{T}(\mathtt{X}) = \mathtt{[bool]} \to \mathtt{nat}$, $\mathtt{s} :: \mathtt{nat} \to \mathtt{nat}$ in $PA(\hat{T}, \mathtt{s\ N})$ and $mgu(\mathtt{[bool]} \to \mathtt{nat} \approx \mathtt{nat} \to \mathtt{nat}) = \mathbf{FAIL}$.

   In the case of G, dynamic type checking does not prevent a type error. One gets: $T(\mathtt{X}) = \alpha \to \mathtt{nat}$, $\mathtt{s} :: \mathtt{nat} \to \mathtt{nat}$ and $mgu(\alpha \to \mathtt{nat} \approx \mathtt{nat} \to \mathtt{nat}) = \{\alpha \mapsto \mathtt{nat}\}$ and hence $G \Vdash_{\mathbf{BD},\mathrm{wrong}} G'$, where

$$G' = \emptyset\ \square\ \mathtt{Y} \bowtie \mathtt{N},\ \mathtt{negate\ Y} \bowtie \mathtt{Z}\ \square\ \mathtt{X} \approx \mathtt{s}\ \square\ \alpha \approx \mathtt{nat}\ \square \\ \{\mathtt{X} :: \mathtt{nat} \to \mathtt{nat},\ \mathtt{Y} :: \mathtt{[bool]},\ \mathtt{Z} :: \mathtt{[bool]},\ \mathtt{N} :: \mathtt{nat}\}$$

   Note that G' is ill-typed, although it admits an untyped solution: $\sigma_d = \{\mathtt{X} \mapsto \mathtt{s}, \mathtt{N} \mapsto \mathtt{[\,]}, \mathtt{Y} \mapsto \mathtt{[\,]}, \mathtt{Z} \mapsto \mathtt{[\,]}\}$.

2. Opaque decomposition escapes from dynamic type checking. This is not possible for transformation **BD**. Since $X\ \overline{a_k} \asymp s\ \overline{b_k}$ is well-typed w.r.t. $\hat{T}$, the type $\hat{T}(X)$ and the type annotation of s in $PA(\hat{T}, s\ \overline{b_k})$ must have the form $\overline{\mu_k} \to \mu$ and $\overline{\nu_k} \to \mu$, respectively, with $\overline{\mu_k}$ and $\overline{\nu_k}$ such that $\hat{T} \vdash_{WT} \overline{a_k} :: \overline{\mu_k}$ and $\hat{T} \vdash_{WT} \overline{b_k} :: \overline{\nu_k}$. Moreover, dynamic type checking unifies $\overline{\mu_k} \to \mu$ and $\overline{\nu_k} \to \mu$ as part of the **BD** transformation. Therefore, $\overline{a_k}$ and $\overline{b_k}$ cannot become ill-typed in the new goal.

Example 13 (CLNC Transformation IM)

1. Type-closure and dynamic checking are necessary. Consider $G \neq \hat{G}$, given as:

$$G = \emptyset \ \square\ \mathtt{X\ Y} \bowtie \mathtt{cons\ (head\ [z])\ L},\ \mathtt{negate\ Y} \bowtie \mathtt{Z} \ \square\ \emptyset \ \square\ \emptyset \ \square \\ \{\mathtt{X} :: \alpha \to \mathtt{[nat]},\ \mathtt{Y} :: \mathtt{[bool]},\ \mathtt{L} :: \mathtt{[nat]},\ \mathtt{Z} :: \mathtt{[bool]}\}$$

$$\hat{G} = \emptyset \ \square\ \mathtt{X\ Y} \bowtie \mathtt{cons\ (head\ [z])\ L},\ \mathtt{negate\ Y} \bowtie \mathtt{Z} \ \square\ \emptyset \ \square \\ \alpha \approx \mathtt{[bool]} \ \square\ \{\mathtt{X} :: \mathtt{[bool]} \to \mathtt{[nat]},\ \mathtt{Y} :: \mathtt{[bool]}, \\ \mathtt{L} :: \mathtt{[nat]},\ \mathtt{Z} :: \mathtt{[bool]}\}$$

Dynamic type checking prevents the application of **IM** to $\hat{G}$ with binding $\{\mathtt{X} \mapsto \mathtt{cons\ A}\}$, since $\hat{T}(\mathtt{X}) = \mathtt{[bool]} \to \mathtt{[nat]}$, $\mathtt{cons\ A} :: \mathtt{[}\alpha_1\mathtt{]} \to \mathtt{[}\alpha_1\mathtt{]}$ and $mgu(\mathtt{[bool]} \to \mathtt{[nat]} \approx \mathtt{[}\alpha_1\mathtt{]} \to \mathtt{[}\alpha_1\mathtt{]}) = \mathbf{FAIL}$.

In the case of G, dynamic type checking does not prevent a type error. One gets: $T(\mathtt{X}) = \alpha \to \mathtt{[nat]}$, $\mathtt{cons\ A} :: \mathtt{[}\alpha_1\mathtt{]} \to \mathtt{[}\alpha_1\mathtt{]}$, $mgu(\alpha \to \mathtt{[nat]} \approx \mathtt{[}\alpha_1\mathtt{]} \to \mathtt{[}\alpha_1\mathtt{]}) = \{\alpha \mapsto \mathtt{[nat]}, \alpha_1 \mapsto \mathtt{nat}\}$ and hence $G \Vdash_{\mathbf{IM},\mathrm{wrong}} G'$, where

$$G' = \emptyset \ \square\ \mathtt{A} \bowtie \mathtt{head\ [z]},\ \mathtt{Y} \bowtie \mathtt{L},\ \mathtt{negate\ Y} \bowtie \mathtt{Z} \ \square\ \mathtt{X} \approx \mathtt{cons\ A} \ \square \\ \alpha \approx \mathtt{[nat]},\ \alpha_1 \approx \mathtt{nat} \ \square\ \{\mathtt{X} :: \mathtt{[nat]} \to \mathtt{[nat]},\ \mathtt{Y} :: \mathtt{[bool]}, \\ \mathtt{L} :: \mathtt{[nat]},\ \mathtt{Z} :: \mathtt{[bool]},\ \mathtt{A} :: \mathtt{nat}\}$$

Note that G' is ill-typed, although it admits a solution: $\sigma_d = \{\mathtt{X} \mapsto \mathtt{cons\ z}, \mathtt{A} \mapsto \mathtt{z}, \mathtt{Y} \mapsto \mathtt{[\,]}, \mathtt{L} \mapsto \mathtt{[\,]}, \mathtt{Z} \mapsto \mathtt{[\,]}\}$.

2. Opaque decomposition escapes from dynamic type checking. Consider $G = \hat{G}$, given as:

$$G = \emptyset \ \square\ \mathtt{X\ (tail\ [z])} \bowtie \mathtt{third\ one\ (tail\ [true])} \ \square\ \emptyset \ \square\ \emptyset \ \square \\ \{\mathtt{X} :: \mathtt{[nat]} \to \beta \to \beta\}$$

Dynamic type checking does not prevent the application of **IM** to G with binding $\{\mathtt{X} \mapsto \mathtt{third\ A}\}$. In fact $T(\mathtt{X}) = \mathtt{[nat]} \to \beta \to \beta$, $\mathtt{third\ A} :: \alpha_1 \to \beta_1 \to \beta_1$, $mgu(\mathtt{[nat]} \to \beta \to \beta \approx \alpha_1 \to \beta_1 \to \beta_1) = \{\alpha_1 \mapsto \mathtt{[nat]}, \beta_1 \mapsto \beta\}$ and therefore:

$$G \Vdash_{\mathbf{IM}} G' = \emptyset \ \square\ \mathtt{A} \bowtie \mathtt{one},\ \mathtt{tail\ [z]} \bowtie \mathtt{tail\ [true]} \ \square\ \mathtt{X} \approx \mathtt{third\ A} \\ \square\ \alpha_1 \approx \mathtt{[nat]},\ \beta_1 \approx \beta \ \square\ \{\mathtt{X} :: \mathtt{[nat]} \to \beta \to \beta, \mathtt{A} :: \delta\}$$

The step $G \Vdash_{IM} G'$ involves opaque decomposition, and G' turns out to be ill-typed. Nevertheless G' admits the solution $\sigma_d = \{\mathtt{A} \mapsto \mathtt{s\ z}, \mathtt{X} \mapsto \mathtt{third\ (s\ z)}\}$.

Example 14 (CLNC Transformation GN)
Type-closure and dynamic checking are necessary. Consider $G \neq \hat{G}$, given as:

$$G = \emptyset \ \square\ \mathtt{F\ X} \bowtie \mathtt{true} \ \square\ \emptyset \ \square\ \emptyset \ \square\ \{\mathtt{X} :: \alpha_1,\ \mathtt{F} :: \alpha_1 \to \alpha\}$$
$$\hat{G} = \emptyset \ \square\ \mathtt{F\ X} \bowtie \mathtt{true} \ \square\ \emptyset \ \square\ \alpha \approx \mathtt{bool} \ \square\ \{\mathtt{X} :: \alpha_1,\ \mathtt{F} :: \alpha_1 \to \mathtt{bool}\}$$

Dynamic type checking prevents the application of **GN** to $\hat{G}$ with binding $\{\mathtt{F} \mapsto \mathtt{plus\ z}\}$, since $\hat{T}(\mathtt{F}) = \alpha_1 \to \mathtt{bool}$, $\mathtt{plus\ z} :: \mathtt{nat} \to \mathtt{nat}$ and $mgu(\alpha_1 \to \mathtt{bool} \approx \mathtt{nat} \to \mathtt{nat}) = \mathbf{FAIL}$.

In the case of G, dynamic type checking does not prevent a type error. One gets: $T(\mathtt{F}) = \alpha_1 \to \alpha$, $\mathtt{plus\ z} :: \mathtt{nat} \to \mathtt{nat}$, $mgu(\alpha_1 \to \alpha \approx \mathtt{nat} \to \mathtt{nat}) = \{\alpha \mapsto \mathtt{nat}, \alpha_1 \mapsto \mathtt{nat}\}$ and hence $G \Vdash_{\mathbf{GN},\mathrm{wrong}} G'$, where

$$G' = \mathtt{X} \to \mathtt{N} \ \square\ \mathtt{N} \bowtie \mathtt{true} \ \square\ \mathtt{F} \approx \mathtt{plus\ z} \ \square\ \alpha \approx \mathtt{nat},\ \alpha_1 \approx \mathtt{nat} \ \square$$
$$\{\mathtt{F} :: \mathtt{nat} \to \mathtt{nat},\ \mathtt{X} :: \mathtt{nat},\ \mathtt{N} :: \mathtt{nat}\}$$

Note that G' is ill-typed, although it admits a solution: $\sigma_d = \{\mathtt{F} \mapsto \mathtt{plus\ z}, \mathtt{X} \mapsto \mathtt{true}, \mathtt{N} \mapsto \mathtt{true}\}$.

Example 15 (CLNC Transformation GD)

1. Type-closure and dynamic checking are necessary. Consider $G \neq \hat{G}$, given as:

$$G = \emptyset \ \square\ \mathtt{X\ (tail\ [z])} \bowtie \mathtt{Y\ (tail\ [true])} \ \square\ \emptyset \ \square\ \emptyset \ \square$$
$$\{\mathtt{X} :: \alpha_1 \to \beta,\ \mathtt{Y} :: \alpha_2 \to \beta\}$$
$$\hat{G} = \emptyset \ \square\ \mathtt{X\ (tail\ [z])} \bowtie \mathtt{Y\ (tail\ [true])} \ \square\ \emptyset \ \square\ \alpha_1 \approx \mathtt{[nat]},$$
$$\alpha_2 \to \mathtt{[bool]} \ \square\ \{\mathtt{X} :: \mathtt{[nat]} \to \beta, \mathtt{Y} :: \mathtt{[bool]} \to \beta\}$$

   Dynamic type checking prevents the application of **GD** to $\hat{G}$ with binding $\{\mathtt{X} \mapsto \mathtt{cons\ A}, \mathtt{Y} \mapsto \mathtt{cons\ B}\}$, since $\hat{T}(\mathtt{X}) = \mathtt{[nat]} \to \beta$, $\hat{T}(\mathtt{Y}) = \mathtt{[bool]} \to \beta$, $\mathtt{cons\ A} :: [\beta_1] \to [\beta_1]$, $\mathtt{cons\ B} :: [\beta_2] \to [\beta_2]$ and $mgu(\mathtt{[nat]} \to \beta \approx [\beta_1] \to [\beta_1],\ \mathtt{[bool]} \to \beta \approx [\beta_2] \to [\beta_2]) = \mathbf{FAIL}$.

In the case of G, dynamic type checking does not prevent a type error. One gets: $T(\mathtt{X}) = \alpha_1 \to \beta$, $T(\mathtt{Y}) = \alpha_2 \to \beta$, $\mathtt{cons\ A} :: [\beta_1] \to [\beta_1]$, $\mathtt{cons\ B} :: [\beta_2] \to [\beta_2]$ and $mgu(\alpha_1 \to \beta \approx [\beta_1] \to [\beta_1],\ \alpha_2 \to \beta \approx [\beta_2] \to [\beta_2]) = \{\alpha_1 \mapsto [\beta_1], \alpha_2 \mapsto [\beta_1], \beta \mapsto [\beta_1], \beta_2 \mapsto \beta_1\}$ and hence $G \Vdash_{\mathbf{GD},\mathrm{wrong}} G'$:

$$G' = \emptyset \ \square\ \mathtt{A} \bowtie \mathtt{B},\ \mathtt{tail}\,[\mathtt{z}] \bowtie \mathtt{tail}\ [\mathtt{true}] \ \square\ \mathtt{X} \approx \mathtt{cons\ A},$$
$$\mathtt{Y} \approx \mathtt{cons\ B} \ \square\ \alpha_1 \approx [\beta_1],\ \alpha_2 \approx [\beta_1],\ \beta \approx [\beta_1],\ \beta_2 \approx \beta_1 \ \square$$
$$\{\mathtt{X} :: [\beta_1] \to [\beta_1],\ \mathtt{Y} :: [\beta_1] \to [\beta_1],\ \mathtt{A} :: \beta_1,\ \mathtt{B} :: \beta_1\}$$

Note that G' is ill-typed, although it admits a solution: $\sigma_d = \{\mathtt{X} \mapsto \mathtt{cons\ A}, \mathtt{Y} \mapsto \mathtt{cons\ A}, \mathtt{B} \mapsto \mathtt{A}\}$.

2. Opaque decomposition escapes from dynamic type checking. Consider $G = \hat{G}$, given as:

$$G = \emptyset \ \square\ \mathtt{X\ (tail\ [z])} \bowtie \mathtt{Y\ (tail\ [true])} \ \square\ \emptyset \ \square\ \emptyset \ \square$$
$$\{\mathtt{X} :: \mathtt{[nat]} \to \beta,\ \mathtt{Y} :: \mathtt{[bool]} \to \beta\}$$

Dynamic type checking does not prevent the application of **GD** to G with binding $\{\mathtt{X} \mapsto \mathtt{snd}, \mathtt{Y} \mapsto \mathtt{snd}\}$. In fact $T(\mathtt{X}) = \mathtt{[nat]} \to \beta$, $T(\mathtt{Y}) = \mathtt{[bool]} \to \beta$, $\mathtt{snd} :: \alpha_1 \to \beta_1 \to \beta_1$, $\mathtt{snd} :: \alpha_2 \to \beta_2 \to \beta_2$, $mgu(\mathtt{[nat]} \to \beta \approx \alpha_1 \to \beta_1 \to \beta_1,\ \mathtt{[bool]} \to \beta \approx \alpha_2 \to \beta_2 \to \beta_2) = \{\alpha_1 \mapsto \mathtt{[nat]}, \alpha_2 \mapsto \mathtt{[bool]}, \beta \mapsto (\beta_1 \to \beta_1), \beta_2 \mapsto \beta_1\}$ and therefore:

$$G \Vdash_{\mathbf{GD}} G' = \emptyset \ \square\ \mathtt{tail}\,[\mathtt{z}] \bowtie \mathtt{tail\ [true]} \ \square\ \mathtt{X} \approx \mathtt{snd},\ \mathtt{Y} \approx \mathtt{snd} \ \square$$
$$\alpha_1 \approx \mathtt{[nat]},\ \alpha_2 \approx \mathtt{[bool]},\ \beta \approx (\beta_1 \to \beta_1),\ \beta_2 \approx \beta_1 \ \square$$
$$\{\mathtt{X} :: \mathtt{[nat]} \to \beta_1 \to \beta_1, \mathtt{Y} :: \mathtt{[bool]} \to \beta_1 \to \beta_1\}$$

The step $G \Vdash_{\mathbf{GD}} G'$ involves opaque decomposition, and G' turns out to be ill-typed. Nevertheless G' admits the solution $\sigma_d = \{\mathtt{X} \mapsto \mathtt{snd}, \mathtt{Y} \mapsto \mathtt{snd}\}$.

Example 16 (CLNC Transformation OB)
Type-closure and dynamic checking are necessary. Consider $G \neq \hat{G}$, given as:

$$G = \mathtt{X\ Y} \to \mathtt{s\ N} \ \square\ \mathtt{negate\ Y} \bowtie \mathtt{Z} \ \square\ \emptyset \ \square\ \emptyset \ \square\ \{\mathtt{X} :: \alpha \to \mathtt{nat},$$
$$\mathtt{Y} :: \mathtt{[bool]},\ \mathtt{Z} :: \mathtt{[bool]},\ \mathtt{N} :: \mathtt{nat}\}$$
$$\hat{G} = \mathtt{X\ Y} \to \mathtt{s\ N} \ \square\ \mathtt{negate\ Y} \bowtie \mathtt{Z} \ \square\ \emptyset \ \square\ \alpha \approx \mathtt{[bool]} \ \square$$
$$\{\mathtt{X} :: \mathtt{[bool]} \to \mathtt{nat},\ \mathtt{Y} :: \mathtt{[bool]},\ \mathtt{Z} :: \mathtt{[bool]},\ \mathtt{N} :: \mathtt{nat}\}$$

Dynamic type checking prevents the application of **OB** to $\hat{G}$ with binding $\{\mathtt{X} \mapsto \mathtt{s}\}$, since $\hat{T}(\mathtt{X}) = \mathtt{[bool]} \to \mathtt{nat}$, $\mathtt{s} :: \mathtt{nat} \to \mathtt{nat}$ and $mgu(\mathtt{[bool]} \to \mathtt{nat} \approx \mathtt{nat} \to \mathtt{nat}) = \mathbf{FAIL}$.

In the case of G, dynamic type checking does not prevent a type error. One gets: $T(\mathtt{X}) = \alpha \to \mathtt{nat}$, $\mathtt{s} :: \mathtt{nat} \to \mathtt{nat}$ and $mgu(\alpha \to \mathtt{nat} \approx \mathtt{nat} \to \mathtt{nat}) = \{\alpha \mapsto \mathtt{nat}\}$ and hence $G \vdash_{\mathbf{OB},\text{wrong}} G'$, where

$$G' = \mathtt{Y} \to \mathtt{N} \ \square\ \mathtt{negate\ Y} \bowtie \mathtt{Z} \ \square\ \mathtt{X} \approx \mathtt{s} \ \square\ \alpha \approx \mathtt{nat} \ \square\ \{\mathtt{X} :: \mathtt{nat} \to \mathtt{nat},\ \mathtt{Y} :: \mathtt{[bool]},\ \mathtt{Z} :: \mathtt{[bool]},\ \mathtt{N} :: \mathtt{nat}\}$$

Note that G' is ill-typed, although it admits a solution: $\sigma_d = \{\mathtt{X} \mapsto \mathtt{s}, \mathtt{Y} \mapsto \mathtt{[]}, \mathtt{Z} \mapsto \mathtt{[]}, \mathtt{N} \mapsto \mathtt{[]}\}$.

Example 17 (CLNC Transformation OGN)
Type-closure and dynamic checking are necessary. Consider $G \neq \hat{G}$, given as:

$$G = \mathtt{F\ X} \to \mathtt{true} \ \square\ \emptyset \ \square\ \emptyset \ \square\ \emptyset \ \square\ \{\mathtt{X} :: \alpha_1,\ \mathtt{F} :: \alpha_1 \to \alpha\}$$
$$\hat{G} = \mathtt{F\ X} \to \mathtt{true} \ \square\ \emptyset \ \square\ \emptyset \ \square\ \alpha \approx \mathtt{bool} \ \square\ \{\mathtt{X} :: \alpha_1,\ \mathtt{F} :: \alpha_1 \to \mathtt{bool}\}$$

Dynamic type checking prevents the application of **OGN** to G with binding $\{\mathtt{F} \mapsto \mathtt{plus\ z}\}$, since $\hat{T}(\mathtt{F}) = \alpha_1 \to \mathtt{bool}$, $\mathtt{plus\ z} :: \mathtt{nat} \to \mathtt{nat}$ and $mgu(\alpha_1 \to \mathtt{bool} \approx \mathtt{nat} \to \mathtt{nat}) = \mathbf{FAIL}$.

In the case of G, dynamic type checking does not prevent a type error. One gets: $T(\mathtt{F}) = \alpha_1 \to \alpha$, $\mathtt{plus\ z} :: \mathtt{nat} \to \mathtt{nat}$, $mgu(\alpha_1 \to \alpha \approx \mathtt{nat} \to \mathtt{nat}) = \{\alpha \mapsto \mathtt{nat}, \alpha_1 \mapsto \mathtt{nat}\}$ and hence $G \vdash_{\mathbf{OGN},\text{wrong}} G'$, where

$$G' = \mathtt{X} \to \mathtt{N},\ \mathtt{N} \to \mathtt{true} \ \square\ \emptyset \ \square\ \mathtt{F} \approx \mathtt{plus\ z} \ \square\ \alpha \approx \mathtt{nat},\ \alpha_1 \approx \mathtt{nat} \ \square\ \{\mathtt{F} :: \mathtt{nat} \to \mathtt{nat},\ \mathtt{X} :: \mathtt{nat},\ \mathtt{N} :: \mathtt{nat}\}$$

Note that G' is ill-typed, although it admits a solution: $\sigma_d = \{\mathtt{F} \mapsto \mathtt{plus\ z}, \mathtt{X} \mapsto \mathtt{true}, \mathtt{N} \mapsto \mathtt{true}\}$.

Example 18 (CLNC Transformation OGD)
Type-closure and dynamic checking are necessary. Consider $G \neq \hat{G}$, given as:

$$G = \mathtt{X\ (tail\ [z])} \to \mathtt{Y} \ \square\ \mathtt{Y} \bowtie \mathtt{[Z|tail\ [true]]} \ \square\ \emptyset \ \square\ \emptyset \ \square\ \{\mathtt{X} :: \alpha \to \mathtt{[}\beta\mathtt{]},\ \mathtt{Y} :: \mathtt{[}\beta\mathtt{]},\ \mathtt{Z} :: \beta\}$$
$$\hat{G} = \mathtt{X\ (tail\ [z])} \to \mathtt{Y} \ \square\ \mathtt{Y} \bowtie \mathtt{[Z|tail\ [true]]} \ \square\ \emptyset \ \square\ \alpha \approx \mathtt{[nat]},\ \beta \approx \mathtt{bool} \ \square\ \{\mathtt{X} :: \mathtt{[nat]} \to \mathtt{[bool]},\ \mathtt{Y} :: \mathtt{[bool]},\ \mathtt{Z} :: \mathtt{bool}\}$$

Dynamic type checking prevents the application of **OGD** to $\hat{G}$ with binding $\{\mathtt{X} \mapsto \mathtt{cons\,A}\}$, since $\hat{T}(\mathtt{X}) = \mathtt{[nat]} \to \mathtt{[bool]}$, $\mathtt{cons\,A} :: \mathtt{[}\alpha_1\mathtt{]} \to \mathtt{[}\alpha_1\mathtt{]}$, $\hat{T}(\mathtt{Y}) = \mathtt{[bool]}$, $\mathtt{cons\,A\,B} :: \mathtt{[}\alpha_2\mathtt{]}$ and $mgu(\mathtt{[nat]} \to \mathtt{[bool]} \approx \mathtt{[}\alpha_1\mathtt{]} \to \mathtt{[}\alpha_1\mathtt{]},\ \mathtt{[bool]} \approx \mathtt{[}\alpha_2\mathtt{]}) = \mathbf{FAIL}$.

In the case of G, dynamic type checking does not prevent a type error. One gets: $T(\mathtt{X}) = \alpha \to \mathtt{[}\beta\mathtt{]}$, $\mathtt{cons\,A} :: \mathtt{[}\alpha_1\mathtt{]} \to \mathtt{[}\alpha_1\mathtt{]}$, $T(\mathtt{Y}) = \mathtt{[}\beta\mathtt{]}$, $\mathtt{cons\,A\,B} :: \mathtt{[}\alpha_2\mathtt{]}$, $mgu(\alpha \to \mathtt{[}\beta\mathtt{]} \approx \mathtt{[}\alpha_1\mathtt{]} \to \mathtt{[}\alpha_1\mathtt{]},\ \mathtt{[}\beta\mathtt{]} \approx \mathtt{[}\alpha_2\mathtt{]}) = \{\alpha \mapsto \mathtt{[}\beta\mathtt{]}, \alpha_1 \mapsto \beta, \alpha_2 \mapsto \beta\}$ and hence $G \Vdash_{\mathbf{OGD},\mathrm{wrong}} G'$, where

$$G' = \mathtt{tail}\,[\mathtt{z}] \to \mathtt{B} \,\square\, \mathtt{[A|B]} \bowtie \mathtt{[Z|tail\ [true]]} \,\square\, \mathtt{X} \approx \mathtt{cons\ A} \,\square\,$$
$$\alpha \approx \mathtt{[}\beta\mathtt{]},\ \alpha_1 \approx \beta,\ \alpha_2 \approx \beta \,\square\, \{\mathtt{X} :: \mathtt{[}\beta\mathtt{]} \to \mathtt{[}\beta\mathtt{]},\ \mathtt{Y} :: \mathtt{[}\beta\mathtt{]},\ \mathtt{Z} :: \beta,$$
$$\mathtt{A} :: \beta,\ \mathtt{B} :: \mathtt{[}\beta\mathtt{]}\}$$

Note that G' is ill-typed, although it admits a solution: $\sigma_d = \{\mathtt{X} \mapsto \mathtt{cons\,Z}, \mathtt{A} \mapsto \mathtt{Z}, \mathtt{B} \mapsto \mathtt{[\,]}\}$.

Examples 11–18 illustrate situations where a type error leads to an ill-typed goal which has nevertheless some solutions, in the sense of Definition 12, items 1, 2. Of course, there are also cases where a type error leads to an unsolvable goal.

4.3 Soundness and Completeness

In this final subsection we present soundness and completeness results for CNLC. The main results are Theorem 5 and Theorem 6, corresponding to Theorem 2 and Theorem 3 in [20]. After the publication of [20], where no proofs were included, we unfortunately discovered some mistakes in our handwritten proofs for these results. Regarding the Soundness Theorem 2 from [20], we missed to recognize that a well-typed solved goal must be type-closed before extracting a solution. Moreover, we did not notice that the empty type environment in the solution extracted from a solved goal must be extended to a generally non-empty environment, in order to obtain a computed answer for the initial goal. With respect to the Completeness Theorem 3 in [20], we failed to recognize that this result should be stated on the basis of solutions for the type-closures of the well-typed goals under consideration, rather than for the goals themselves. This is relevant because of the difference between the two sets of solutions $WTSol(\hat{G})$ and $WTSol(G)$ in the case that G is well-typed, but not type-closed; see Lemma 14. Presently we have found proofs for revised formulations of both theorems.

We are interested in well-typed goals G for a well-typed program $\mathcal{P}$. We write T_G for the type environment of G, and we use the notation $(R, \theta) \in_{ex} WTSol(G)$ to indicate the existence of some $(R', \theta') \in WTSol(G)$ such that θ and θ' can differ only over existential variables. Remember that initial goals are of the form $G_0 = \emptyset \sqcap C \sqcap \emptyset \sqcap \emptyset \sqcap T_0$ and include no existential variables, while solved goals have the form $G_n = \emptyset \sqcap \emptyset \sqcap S_d \sqcap S_t \sqcap T_n$. As defined at the beginning of Subsection 4.2, the answer computed by a CLNC derivation with initial goal G_0 and final solved goal G_n is $(T_n\hat{\sigma}_t \backslash T_0\hat{\sigma}_t, (\hat{\sigma}_t, \sigma_d))$, where the type substitution $\hat{\sigma}_t$ comes from the type-closure of G_n. All these assumptions and notational conventions will be kept in the rest of this subsection.

Now we are ready to discuss soundness. The following lemma (proved in the Appendix) ensures the correctness of computed answers, extracted from solved goals.

Lemma 15 (Computed Answers)
Assume a well-typed solved goal $G = \emptyset \sqcap \emptyset \sqcap S_d \sqcap S_t \sqcap T$ with type-closure $\hat{G} = \emptyset \sqcap \emptyset \sqcap S_d \sqcap \hat{S}_t \sqcap \hat{T}$. Then the computed answer $(\emptyset, (\hat{\sigma}_t, \sigma_d))$ is a well-typed solution of both G and $\hat{G}$. □

Regarding soundness of one single CLNC step, we have the following lemma. A proof is given in the Appendix.

Lemma 16 (One-step Soundness)
Assume a well-typed goal G, for a well-typed program $\mathcal{P}$. Then:

1. *If $G \vdash_{CLNC} \textbf{\textit{FAIL}}$, then $Sol(G) = \emptyset$.*
2. *If $G \vdash_{CLNC} G'$, then G' is an admissible goal, and every $(R, \theta) \in WTSol(G')$ verifies $((T_{G'}\theta_t \backslash T_G\theta_t) \cup R, \theta) \in_{ex} WTSol(G)$. Moreover, G' is again well-typed, unless the CLNC step from G to G' has performed an opaque decomposition.* □

Our desired soundness result follows from Lemma 16:

Theorem 5 (Soundness)
Consider a well-typed program $\mathcal{P}$ and a CLNC derivation $G_0 \vdash_{CLNC} \cdots \vdash_{CLNC} G_n$, where G_0 is well-typed and type-closed initial goal, G_n is a solved goal, and all the steps are transparent. Then all the goals are well-typed and the computed answer is correct, i.e. $(T_n\hat{\sigma}_t \setminus T_0\hat{\sigma}_t, (\hat{\sigma}_t, \sigma_d)) \in WTSol(G_0)$.

Proof: Due to Lemma 15, we know $(\emptyset, (\hat{\sigma}_t, \sigma_d)) \in \mathit{WTSol}(G_n)$. Moreover, the type environments of the goals along the CLNC derivation, affected by $\hat{\sigma}_t$, form an increasing chain $T_0\hat{\sigma}_t \subseteq \cdots \subseteq T_n\hat{\sigma}_t$. Therefore, by reiterated application of Lemma 16, we can obtain $(T_n\hat{\sigma}_t \setminus T_0\hat{\sigma}_t, (\hat{\sigma}_t, \sigma_d)) \in_{ex} \mathit{WTSol}(G_0)$. Since G_0 includes no existential variables, the desired conclusion follows. □

The rôle of the type environment $(T_n\hat{\sigma}_t \setminus T_0\hat{\sigma}_t)$ in a computed answer is to provide type assumptions for the new variables occurring in the range of the computed data substitution. This can be seen in Example 9(b). An additional illustration is provided by the next example, based on the program from Example 2.

Example 19 (Computed Answer with Non-empty Type Environment)

$$\hat{G}_0 = G_0 = \emptyset \;\square\; \underline{\mathtt{pp\,N} \bowtie \mathtt{true}} \;\square\; \emptyset \;\square\; \emptyset \;\square\; \{\mathtt{N} :: \mathtt{nat}\} \quad \vdash_{\mathbf{NR1}}$$

$$G_1 = \underline{\mathtt{N} \to \mathtt{s\,X}} \;\square\; \mathtt{p\,X} \bowtie \mathtt{true} \;\square\; \emptyset \;\square\; \emptyset \;\square\; \{\mathtt{N}, \mathtt{X} :: \mathtt{nat}\} \quad \vdash_{\mathbf{OB}}$$

$$G_2 = \emptyset \;\square\; \underline{\mathtt{p\,X} \bowtie \mathtt{true}} \;\square\; \mathtt{N} \approx \mathtt{s\,X} \;\square\; \emptyset \;\square\; \{\mathtt{N}, \mathtt{X} :: \mathtt{nat}\} \quad \vdash_{\mathbf{NR1}}$$

$$G_3 = \underline{\mathtt{X} \to \mathtt{s\,Y}} \;\square\; \mathtt{true} \bowtie \mathtt{true} \;\square\; \mathtt{N} \approx \mathtt{s\,X} \;\square\; \emptyset \;\square\; \{\mathtt{N}, \mathtt{X} :: \mathtt{nat}\} \quad \vdash_{\mathbf{OB}}$$

$$G_4 = \emptyset \;\square\; \underline{\mathtt{true} \bowtie \mathtt{true}} \;\square\; \mathtt{N} \approx \mathtt{s\,(s\,Y)}, \mathtt{X} \approx \mathtt{s\,Y} \;\square\; \emptyset \;\square\; \{\mathtt{N}, \mathtt{X}, \mathtt{Y} :: \mathtt{nat}\} \vdash_{\mathbf{DC1}}$$

$$G_5 = \emptyset \;\square\; \emptyset \;\square\; \mathtt{N} \approx \mathtt{s\,(s\,Y)}, \mathtt{X} \approx \mathtt{s\,Y} \;\square\; \emptyset \;\square\; \{\mathtt{N}, \mathtt{X}, \mathtt{Y} :: \mathtt{nat}\}$$

$G_5 = \hat{G}_5$ yields the computed answer $(T_{G_5}\hat{\sigma}_t \setminus T_{G_0}\hat{\sigma}_t, \sigma)$ with $T_{G_5}\hat{\sigma}_t \setminus T_{G_0}\hat{\sigma}_t = \{\mathtt{X}, \mathtt{Y} :: \mathtt{nat}\}$, $\sigma_t = \emptyset$ and $\sigma_d = \{\mathtt{N} \mapsto \mathtt{s\,(s\,Y)}, \mathtt{X} \mapsto \mathtt{s\,Y}\}$.

This computed answer is a well-typed solution of G_0, as predicted by Theorem 5. The type environment $(T_{G_5}\hat{\sigma}_t \setminus T_{G_0}\hat{\sigma}_t)$ is necessary, since $(\emptyset, \sigma) \notin \mathit{WTSol}(G_0)$.

In order to prove completeness of CLNC, we use a well-founded ordering over witnesses of solutions, as in [19, 18]. Let $\prec$ be the well-founded multiset ordering for multisets of natural numbers [11, 6]. Then:

Definition 14 (Ordering for Witnesses)
Let $\mathcal{M}$, $\mathcal{M}'$ be finite multisets of (possibly type-annotated) GORC proofs. Let $\mathcal{SM}$, $\mathcal{SM}'$ be the corresponding multisets of natural numbers, obtained

by replacing each GORC proof by its size*, understood as the number of GORC inference steps. Then we define* $\mathcal{M} \lhd \mathcal{M}'$ *iff* $\mathcal{SM} \prec \mathcal{SM}'$.

The next result, also proved in the Appendix, guarantees that CLNC transformations can be chosen to make progress according to a given type-annotated witness.

Lemma 17 (One-step Completeness)
Assume a well-typed goal G*, for a well-typed program* $\mathcal{P}$*, as well as a solution* $(R, \theta) \in TASol(\hat{G})$ *with type-annotated witness* $\mathcal{M}$*. If* G *is not yet solved, there is some well-typed goal* G' *such that* $G \vdash_{CLNC} G'$ *and* $(R, \theta) \in_{ex} TASol(\hat{G'})$ *with witness* $\mathcal{M}' \lhd \mathcal{M}$*.* □

In contrast to the formulation of Lemma 2 in [20], the previous lemma assumes well-typed solutions of the *type closures* of the goals under consideration. Omitting the type closures would lead to the following problem: G might be not type-closed, and G' might become type-closed by virtue of one of the CLNC transformations marked with ★. In this case, proving $(R, \theta) \in_{ex} TASol(\hat{G'})$ might be impossible due to item 1 from Lemma 14.

Our desired completeness result can be easily deduced from Lemma 17:

Theorem 6 (Completeness)
Assume a well-typed goal G_0*, for a well-typed program* $\mathcal{P}$*, as well as a type-annotated solution* $(R, \theta) \in TASol(G_0)$*. Then there exists a CLNC derivation* $G_0 \vdash^*_{CLNC} G_n$ *consisting entirely of well-typed goals, where the solved form* G_n *with associated computed answer* $(T_n\hat{\sigma}_t \setminus T_0\hat{\sigma}_t, (\hat{\sigma}_t, \sigma_d))$ *is such that* $(R, \theta) \in_{ex} TASol(\hat{G}_n)$*, which implies* $\hat{\sigma}_t \leq \theta_t[tvar(G_0)]$ *and* $\sigma_d \leq \theta_d[var(G_0)]$.

Proof: Reiterate the application of Lemma 17, starting from $(R, \theta) \in TASol(\hat{G}_0)$, which follows from the hypothesis because G_0 is type-closed. Since the ordering $\lhd$ is well-founded, the reiteration must eventually terminate with a well-typed solved goal $\hat{G}_n$ such that $(R, \theta) \in_{ex} TASol(\hat{G}_n)$. This means the existence of some $(R', \theta') \in TASol(\hat{G}_n)$ such that θ and θ' can differ only over existential variables. Then $\theta'_t = \hat{\sigma}_t\theta'_t$ and $\theta'_d = \sigma_d\theta'_d$ follows from $\theta'_t \in TSol(\hat{S}_t)$ and $\theta'_d \in Sol(S_d)$, respectively. Since G_0 includes no existential variables, we can conclude that $\hat{\sigma}_t \leq \theta_t[tvar(G_0)]$ and $\sigma_d \leq \theta_d[var(G_0)]$ as desired. □

Example 20 below illustrates the need of a type-annotated witness in the hypothesis of Theorem 6. We assume the program from Example 2. The

initial goal G_0 admits the well-typed solution $(\emptyset, (\emptyset, \{\mathtt{X} \mapsto \mathtt{[]}, \mathtt{Y} \mapsto \mathtt{[]}\}))$. Due to opacity of `snd`, no type-annotated witness for this solution exists. In the CLNC derivation shown below, the goal G_1 becomes ill-typed after the first opaque decomposition step. The rest of the CLNC steps do not require any dynamic type checking and can be performed in spite of the fact that the goals are not well-typed. The ill-typed solved goal G_7 obtained at the end provides no well-typed computed answer. Nevertheless, the data substitution σ_d extracted from G_7 subsumes the data part of the initially given solution. This is consistent with the completeness of *untyped* CLNC, as proved in Theorem 4.2 from [19].

Example 20 (A Well-typed Solution Lacking a Type-annotated Witness)

$$\hat{G}_0 = G_0 = \emptyset \;\square\; \underline{\mathtt{snd\ (head\ [X,[true]])} \bowtie \mathtt{snd\ (head\ [Y,[z]])}} \;\square\; \emptyset \;\square\; \emptyset \;\square\; \{\mathtt{X} :: \mathtt{[bool]},\ \mathtt{Y} :: \mathtt{[nat]}\} \;\vdash_{\mathbf{DC1}}$$

$$\hat{G}_1 \neq G_1 = \emptyset \;\square\; \underline{\mathtt{head\ [X,[true]]}} \bowtie \mathtt{head\ [Y,[z]]} \;\square\; \emptyset \;\square\; \emptyset \;\square\; \{\mathtt{X} :: \mathtt{[bool]},\ \mathtt{Y} :: \mathtt{[nat]}\} \;\vdash_{\mathbf{NR1}}$$

$$G_2 = \underline{\mathtt{[X|[[true]]]} \rightarrow \mathtt{[X1|X1s]}} \;\square\; \mathtt{X1} \bowtie \mathtt{head\ [Y,[z]]} \;\square\; \emptyset \;\square\; \emptyset \;\square\; \{\mathtt{X} :: \mathtt{[bool]},\ \mathtt{Y} :: \mathtt{[nat]},\ \mathtt{X1} :: \alpha_1,\ \mathtt{X1s} :: [\alpha_1]\} \;\vdash_{\mathbf{DC2}}$$

$$G_3 = \underline{\mathtt{X} \rightarrow \mathtt{X1},\ \mathtt{[[true]]} \rightarrow \mathtt{X1s}} \;\square\; \mathtt{X1} \bowtie \mathtt{head\ [Y,[z]]} \;\square\; \emptyset \;\square\; \emptyset \;\square\; \{\mathtt{X} :: \mathtt{[bool]},\ \mathtt{Y} :: \mathtt{[nat]},\ \mathtt{X1} :: \alpha_1,\ \mathtt{X1s} :: [\alpha_1]\} \;\vdash_{\mathbf{IB}^2}$$

$$G_4 = \emptyset \;\square\; \mathtt{X} \bowtie \underline{\mathtt{head\ [Y,[z]]}} \;\square\; \emptyset \;\square\; \emptyset \;\square\; \{\mathtt{X} :: \mathtt{[bool]},\ \mathtt{Y} :: \mathtt{[nat]},\ \mathtt{X1} :: \alpha_1,\ \mathtt{X1s} :: [\alpha_1]\} \;\vdash_{\mathbf{NR1}}$$

$$G_5 = \underline{\mathtt{[Y|[[z]]]} \rightarrow \mathtt{[Y1|Y1s]}} \;\square\; \mathtt{X} \bowtie \mathtt{Y1} \;\square\; \emptyset \;\square\; \emptyset \;\square\; \{\mathtt{X} :: \mathtt{[bool]},\ \mathtt{Y} :: \mathtt{[nat]},\ \mathtt{X1} :: \alpha_1,\ \mathtt{X1s} :: [\alpha_1],\ \mathtt{Y1} :: \alpha_2,\ \mathtt{Y1s} :: [\alpha_2]\} \;\vdash_{\mathbf{DC2},\mathbf{IB}^2}$$

$$G_6 = \emptyset \;\square\; \underline{\mathtt{X} \bowtie \mathtt{Y}} \;\square\; \emptyset \;\square\; \emptyset \;\square\; \{\mathtt{X} :: \mathtt{[bool]},\ \mathtt{Y} :: \mathtt{[nat]},\ \mathtt{X1} :: \alpha_1,\ \mathtt{X1s} :: [\alpha_1],\ \mathtt{Y1} :: \alpha_2,\ \mathtt{Y1s} :: [\alpha_2]\} \;\vdash_{\mathbf{BD}}$$

$$G_7 = \emptyset \;\square\; \emptyset \;\square\; \mathtt{X} \approx \mathtt{Y} \;\square\; \emptyset \;\square\; \{\mathtt{X} :: \mathtt{[bool]},\ \mathtt{Y} :: \mathtt{[nat]},\ \mathtt{X1} :: \alpha_1,\ \mathtt{X1s} :: [\alpha_1],\ \mathtt{Y1} :: \alpha_2,\ \mathtt{Y1s} :: [\alpha_2]\}$$

5 Conclusions

We have presented a polymorphic type system which extends a previous approach to HO FLP, based on the rewriting logic CRWL [19]. We have defined a natural class of well-typed programs, and we have extended both the models and the lazy narrowing calculus CLNC from [19] to take types into account. Our logical semantics assigns a meaning both to well-typed and to untyped programs. With respect to lazy narrowing, we have identified two possible sources of run-time type errors in CLNC computations, namely *opaque decompositions* and *ill-typed bindings for HO logic variables*.

Regarding the first problem, we have shown that the eventual occurrence of opaque decomposition steps is undecidable in general, and we have argued that the difficulty is nevertheless bearable from a practical viewpoint. On the other hand, we have proposed dynamic type checking mechanisms to prevent the second problem, causing only a small overhead in computations that involve no use of HO logic variables acting as functions. We have proved that CLNC with dynamic type checking is sound and complete with respect to the logical semantics in a reasonable sense, namely: computed answers are always correct and well-typed unless opaque decomposition has taken place; and correct solutions which have a type-annotated witness are covered by well-typed computed answers.

As future work, we are interested in working out and testing an implementation of dynamic type checking. The $\mathcal{TOY}$ system [32], which works on top on Prolog, is an available possibility. Another interesting approach would be to extend the abstract machine from [27] (whose design supports extensibility and modifiability) with dedicated mechanisms for dynamic type checking.

Acknowledgements We are grateful to two anonymous reviewers for their stimulating criticisms. We also thank our colleagues Miguel Palomino and Eva Ullán, who carefully read an earlier version of this paper and made many useful comments and suggestions.

References

[1] S. Antoy and A. Tolmach. Typed Higher-order Narrowing without Higher-Order Strategies. In *Proc. 4th Int. Symposium on Functional and*

Logic Programming (FLOPS'99), volume 1722 of *LNCS*, pages 335–352. Springer-Verlag, 1999.

[2] K.R. Apt. Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 10, pages 493–574. Elsevier and The MIT Press, 1990.

[3] K.R. Apt and M. Gabbrielli. Declarative Interpretations Reconsidered. In *Proc. Int. Conf. on Logic Programming (ICLP'94)*, The MIT Press, pages 74–89, 1994.

[4] P. Arenas-Sánchez and M. Rodríguez-Artalejo. A Lazy Narrowing Calculus for Functional Logic Programming with algebraic polymorphic types. In *Proc. Int. Symp. on Logic Programming (ILPS'97)*, The MIT Press, pages 53–68, 1997.

[5] P. Arenas-Sánchez and M. Rodríguez-Artalejo. A Semantic Framework for Functional Logic Programming with Algebraic Polymorphic Types. In *Proc. Int. Joint Conference on Theory and Practice of Software Development (TAPSOFT'97)*, volume 1214 of *LNCS*, pages 453–464. Springer-Verlag, 1997.

[6] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[7] W. Chen, M. Kifer, and D.S. Warren. Hilog: A Foundation for Higher-Order Logic Programming. *Journal of Logic Programming*, 15:187–230, 1993.

[8] N. Cutland. *Computability. An Introduction to Recursive Function Theory.* Cambridge University Press, 1980.

[9] L. Damas and R. Milner. Principal Type Schemes for Functional Programs. In *Proc. ACM Symp. on Principles of Programming Languages (POPL'82)*, ACM Press, pages 207–212, 1982.

[10] N. Dershowitz and J.P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 243–320. Elsevier and The MIT Press, 1990.

[11] N. Dershowitz and Z. Manna. Proving Termination with Multiset Orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[12] F. Pfenning (ed.). *Types in Logic Programming*. The MIT Press, 1992.

[13] M. Hanus (ed.). Curry: an Integrated Functional Logic Language, version 0.7.1. Technical report, Universität Kiel, June 2000. Available at `http://www.informatik.uni-kiel.de/curry/`.

[14] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 102(1):86–113, 1993.

[15] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel-LEAF: A Logic plus Functional Language. *Journal of Computer and System Science*, 42(2):139–185, 1991.

[16] J.A. Goguen and J. Meseguer. Models and Equality for Logical Programming. In *Proc. Int. Joint Conference on Theory and Practice of Software Development (TAPSOFT'87)*, volume 250 of *LNCS*, pages 1–22. Springer-Verlag, 1987.

[17] W. Goldfarb. The Undecidibility of the Second-Order Unification Problem. *Theoretical Computer Science*, 13:225–230, 1981.

[18] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An Approach to Declarative Programming Based on a Rewriting Logic. *Journal of Logic Programming*, 40(1):47–87, 1999.

[19] J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. A Higher Order Rewriting Logic for Functional Logic Programming. In *Proc. Int. Conf. on Logic Programming (ICLP'97)*, The MIT Press, pages 153–167, 1997.

[20] J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. Semantics and Types in Functional Logic Programming. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, volume 1722 of *LNCS*, pages 1–20. Springer-Verlag, 1999.

[21] C.A. Gunter and D. Scott. Semantic Domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 633–674. Elsevier and The MIT Press, 1990.

[22] M. Hanus. Horn Clause Programs with Polymorphic Types: Semantics and Resolution. In *Proc. Int. Joint Conf. on Theory and Practice of Software Development (TAPSOFT'89)*, volume 352 of *LNCS*, pages 225–240. Springer-Verlag, 1989.

[23] M. Hanus. Polymorphic Higher-Order Programming in Prolog. In *Proc. Int. Conf. on Logic Programming (ICLP'89)*, The MIT Press, pages 382–397, 1989.

[24] M. Hanus. A Functional and Logic Language with Polymorphic Types. In *Proc. Int. Symp. on Design and Implementation of Symbolic Computation Systems*, volume 429 of *LNCS*, pages 215–224. Springer-Verlag, 1990.

[25] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

[26] S. Hölldobler. *Foundations of Equational Logic Programming*. LNCS. Springer-Verlag, 1989.

[27] M.T. Hortalá-González and E. Ullán. An Abstract Machine Based System for a Lazy Narrowing Calculus. In *Proc. 5th Fuji International Symposium on Functional and Logic Programming (FLOPS'2001)*, volume 2024 of *LNCS*, pages 216–232. Springer-Verlag, 2001.

[28] J. Jaffar, J.L. Lassez, and M.J. Maher. A Theory of Complete Logic Programs with Equality. *Journal of Logic Programming*, 1:211–223, 1984.

[29] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type Reconstruction in the Presence of Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, 1993.

[30] J.W. Klop. Term Rewriting Systems. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 2–116. Oxford University Press, 1992.

[31] K. Kwon, G. Nadathur, and D.S. Wilson. Implementing Polymorphic Typing in a Logic Programming Language. *Computer Languages*, 20(1):25–42, 1994.

[32] F.J. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. 10th Int. Conf. on Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *LNCS*, pages 244–247. Springer-Verlag, 1999. Available at `http://titan.sip.ucm.es/toy/`.

[33] M. Marin, T. Ida, and T. Suzuki. On Reducing the Search Space of Higher-Order Lazy Narrowing. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, volume 1722 of *LNCS*, pages 319–334. Springer-Verlag, 1999.

[34] G. Meyer. Dimensions of Typing in Logic Programming. In *Addendum to the tutorial Types in Logic Programming. Int. Conf. on Logic Programming (ICLP'97)*, 1997.

[35] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.

[36] J.M. Molina-Bravo and E. Pimentel. Modularity in Functional-Logic Programming. In *Proc. Int. Conf. on Logic Programming (ICLP'97)*, The MIT Press, pages 183–197, 1997.

[37] B. Möller. On the Algebraic Specification of Infinite Objects - Ordered and Continuous Models of Algebraic Types. *Acta Informatica*, 22:537–578, 1985.

[38] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The Language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.

[39] A. Mycroft and R.A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[40] G. Nadathur and D. Miller. An Overview of λ-Prolog. In *Proc. Int. Conf. on Logic Programming (ICLP'88)*, The MIT Press, pages 810–827, 1988.

[41] G. Nadathur and F. Pfenning. The Type System of a Higher-Order Logic Programming Language. In F. Pfenning, editor, *Types in Logic Programming*, pages 245–283. The MIT Press, 1992.

[42] K. Nakahara, A. Middeldorp, and T. Ida. A Complete Narrowing Calculus for Higher-Order Functional Logic Programming. In *Proc. Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'95)*, volume 982 of *LNCS*, pages 97–114. Springer-Verlag, 1995.

[43] J. Peterson and K. Hammond (eds.). Report on the Programming Language Haskell 98: A Non-strict, Purely Functional Language. Technical report, February 1999.

[44] C. Prehofer. Higher-Order Narrowing. In *Proc. IEEE Symp. on Logic in Computer Science (LICS'94)*, IEEE Comp. Soc. Press, pages 507–516, 1994.

[45] C. Prehofer. A Call-by-Need Strategy for Higher-Order Functional Logic Programming. In *Proc. Int. Logic Programming Symp. (ILPS'95)*, The MIT Press, pages 147–161, 1995.

[46] C. Prehofer. *Solving Higher Order Equations: From Logic to Programming*. Birkhäuser Verlag, 1998.

[47] M. Rodríguez-Artalejo. Functional and Constraint Logic Programming. In H. Comon, C. Marché, and R. Trainen, editors, *Constraints in Computational Logics*, volume 2002 of *LNCS*, pages 202–270. Springer-Verlag, 2001.

[48] T. Suzuki, K. Nakagawa, and T. Ida. Higher-Order Lazy Narrowing Calculus: A Computation Model for a Higher-Order Functional Logic Language. In *Proc. Int. Conf. on Algebraic and Logic Programming (ALP'97)*, volume 1298 of *LNCS*, pages 99–113. Springer-Verlag, 1997.

[49] D.H.D. Warren. Higher-Order Extensions to Prolog: are they needed? In D. Michie J.E. Hayes and Y.H. Yao, editors, *Machine Intelligence*, volume 10, pages 441–454. Edinburg Univ. Press, 1982.

6 Appendix: Proofs

6.1 Proofs of Results from Subsection 2.2

Proof of Theorem 1: We prove items 1 and 2 simultaneously, by structural induction over e.

Case $X \in DVar$*:* Assume $T(X) = \tau$. Applying **VR** we get $TR(T, X) = (X^\tau, \emptyset)$. Then:

1. For every $\sigma_t \in TSol(\emptyset)$, $T\sigma_t \vdash_{WT} X :: \tau\sigma_t$, due to **VR**.
2. Assume $T \leq T'$ such that $T' \vdash_{WT} X :: \tau'$. Then $T'(X) = \tau'$, and $T' = T\sigma_t$ for some σ_t. This σ_t surely satisfies $\sigma_t \in TSol(\emptyset)$ and $X^\tau\sigma_t = X^{\tau'}$.

Case $h \in DC \cup FS$*:* Consider a fresh variant $(h :: \tau) \in_{var} \Sigma$ of h's principal type declaration. Applying **ID** we get $TR(T, h) = (h^\tau, \emptyset)$. Then:

1. For every $\sigma_t \in TSol(\emptyset)$, $T\sigma_t \vdash_{WT} h :: \tau\sigma_t$, due to **ID**.
2. Assume $T \leq T'$ such that $T' \vdash_{WT} h :: \tau'$. Then $\tau \leq \tau'$, and we can choose σ_t such that $T' = T\sigma_t$, $\tau' = \tau\sigma_t$. This σ_t surely satisfies $\sigma_t \in TSol(\emptyset)$ and $h^\tau\sigma_t = h^{\tau'}$.

Case $(e\, e_1)$*:* Assume $TR(T, e) = (e^\tau, E)$ and $TR(T, e_1) = (e_1^{\tau_1}, E_1)$ with $tvar(E) \cap tvar(E_1) \subseteq ran(T)$. Applying **AP** we obtain $TR(T, (e\ e_1)) = ((e^{\tau_1 \to \gamma}\ e_1^{\tau_1})^\gamma, E \cup E_1 \cup \{\tau \approx \tau_1 \to \gamma\})$, where γ is a fresh type variable not occurring in T, E, E_1. Then:

1. Assume $\sigma_t \in TSol(E \cup E_1 \cup \{\tau \approx \tau_1 \to \gamma\})$. By induction hypothesis for e and e_1, we get $T\sigma_t \vdash_{WT} e :: \tau\sigma_t$ and $T\sigma_t \vdash_{WT} e_1 :: \tau_1\sigma_t$. Moreover, $\tau\sigma_t = \tau_1\sigma_t \to \gamma\sigma_t$ because $\sigma_t \in TSol(\tau \approx \tau_1 \to \gamma)$. Hence, $T\sigma_t \vdash_{WT} (e\, e_1) :: \gamma\sigma_t$ can be derived by applying **AP**.
2. Assume $T \leq T'$ such that $T' \vdash_{WT} (e\, e_1) :: \tau'$. The last step in the derivation must have used **AP**. Therefore, there must be some type τ_1' such that $T' \vdash_{WT} e :: \tau_1' \to \tau'$ and $T' \vdash_{WT} e_1 :: \tau_1'$. By induction hypothesis applied to e and e_1, we can assume $\sigma_t \in TSol(E)$, $\sigma_t^1 \in TSol(E_1)$ such that $T\sigma_t = T'$, $e^\tau\sigma_t = e^{\tau_1' \to \tau'}$, $T\sigma_t^1 = T'$, $e_1^{\tau_1}\sigma_t^1 = e_1^{\tau_1'}$. In particular, we can conclude that $\sigma_t = \sigma_t^1[ran(T)]$. Since γ is a fresh type variable, we can define $\hat{\sigma}_t \in TSub$ such that $\hat{\sigma}_t \restriction ran(T) =$

$\sigma_t \restriction ran(T) = \sigma_t^1 \restriction ran(T)$ and $\hat{\sigma}_t(\gamma) = \tau'$. From this construction it follows that $\hat{\sigma}_t \in TSol(E \cup E_1 \cup \{\tau \approx \tau_1 \to \gamma\})$. In particular, we get: $\tau\hat{\sigma}_t = \tau\sigma_t = \tau_1' \to \tau' = \tau_1\sigma_t^1 \to \gamma\hat{\sigma}_t = (\tau_1 \to \gamma)\hat{\sigma}_t$. Moreover, $e^\tau\hat{\sigma}_t = (e^{\tau_1 \to \gamma}\, e_1^{\tau_1})^\gamma\hat{\sigma}_t = (e^{\tau_1' \to \tau'}\, e_1^{\tau_1'})^{\tau'}$. This concludes the proof. □

Proof of Lemma 2: We reason by induction over the syntactic structure of t. The base cases $t = X$ ($X \in DVar$) and $t = \bot$ are trivial, because the only $t' \sqsupseteq t$ are $\bot$ and t itself. The remaining case is $t = h\,\overline{t}_m$. Due to the assumption $T \vdash_{WT} t :: \tau$, there must be types τ_i ($1 \leq i \leq m$) such that (a) $T \vdash_{WT} e_i :: \tau_i$ for all ($1 \leq i \leq m$) and (b) $\overline{\tau}_m \to \tau$ is an instance of h's principal type. For $t' \sqsupseteq t$ there are only two possibilities: $t' = \bot$ or $t' = h\overline{t'}_m$, where $t_i' \sqsupseteq t_i$ for all ($1 \leq i \leq m$). In the first case, $T \vdash_{WT} \bot :: \tau$ can be derived by **ID**. In the second case, (a) and the induction hypothesis allow us to assume (c) $T \vdash_{WT} t_i' :: \tau_i$ for all ($1 \leq i \leq m$) and then $T \vdash_{WT} h\,\overline{t'}_m :: \tau$ follows from (c) and (b). □

Proof of Lemma 3: We reason by structural induction over e.

Case $\bot$: In this case $e\sigma_d = \bot$ and $T_1 \vdash_{WT} \bot :: \tau\sigma_t$ follows from **ID**, since $(\bot :: \alpha) \in_{var} \Sigma_\bot$.

Case $X \in DVar$: $T_0 \vdash_{WT} X :: \tau$ must be proved by applying **VR**. Hence, $T_0(X) = \tau$ and $T_1 \vdash_{WT} X\sigma_d :: \tau\sigma_t$ holds by hypothesis.

Case $h \in DC \cup FS$: In this case $h\sigma_d = h$ and $T_0 \vdash_{WT} h :: \tau$ must be proved by applying **ID**. Then the principal type of h must be $(h :: \tau_0) \in_{var} \Sigma$ such that $\tau_0 \leq \tau$. Since $\tau_0 \leq \tau \leq \tau\sigma_t$, $T_1 \vdash_{WT} h :: \tau\sigma_t$ follows from **ID**.

Case $(e\,e_1)$: $T_0 \vdash_{WT} (e\,e_1) :: \tau$ must be proved by applying **AP**, having proved previously $T_0 \vdash_{WT} e :: \tau_1 \to \tau$ and $T_0 \vdash_{WT} e_1 :: \tau_1$ for some τ_1. By induction hypothesis for e and e_1, we can assume $T_1 \vdash_{WT} e\sigma_d :: \tau_1\sigma_t \to \tau\sigma_t$ and $T_1 \vdash_{WT} e_1\sigma_d :: \tau_1\sigma_t$. Then $T_1 \vdash_{WT} (e\,e_1)\sigma_d :: \tau\sigma_t$ follows by one application of **AP**. □

Proof of Lemma 4: This follows as a simple corollary from Lemma 3, taking T as T_0, $T\sigma_t$ as T_1, and the identity data substitution id_d as σ_d. □

Proof of Lemma 5: First, note that $e^\tau\sigma$ must be understood as the result of replacing each type τ_0 occurring in e^τ by $\tau_0\sigma_t$ and each variable X^{τ_0} occurring in e^τ by a type-annotated expression $(X\sigma_d)^{\tau_0\sigma_t}$. This leads to a correct type-annotated expression because of Lemmata 1 and 3. □

Proof of Lemma 6: We reason by structural induction over t.

Case $X \in DVar$: In this case, $T_1 \vdash_{WT} X :: \tau$ and $T_2 \vdash_{WT} X :: \tau$ imply $T_1(X) = \tau = T_2(X)$, due to the type inference rule **VR**.

Case $h \in DC \cup FS$: Since $t = h\,\overline{t}_m$ is transparent, h must be m-transparent, and its principal type must be of the form $h :: \overline{\tau}_m \to \tau_0$ with $tvar(\overline{\tau}_m) \subseteq tvar(\tau_0)$. By the assumptions of the lemma and the form of the type derivation rules, there must be types τ_i', τ_i'' $(1 \leq i \leq m)$ such that

(a) $T_1 \vdash_{WT} t_i :: \tau_i'$ for all $(1 \leq i \leq m)$ and $\overline{\tau'}_m \to \tau \leq \overline{\tau}_m \to \tau_0$.

(b) $T_2 \vdash_{WT} t_i :: \tau_i''$ for all $(1 \leq i \leq m)$ and $\overline{\tau''}_m \to \tau \leq \overline{\tau}_m \to \tau_0$.

Since $tvar(\overline{\tau}_m) \subseteq tvar(\tau_0)$, (a) and (b) imply that $\tau_i' = \tau_i''$ for all $(1 \leq i \leq m)$. Then, by the induction hypothesis applied to each t_i, we can conclude that $T_1(X) = T_2(X)$ for all $X \in \bigcup\{var(t_i) \mid 1 \leq i \leq m\} = var(h\,\overline{t}_m)$. □

Proof of Lemma 7: By the assumption of the lemma and the form of the type inference rules, there must be types τ_i, τ_i' $(1 \leq i \leq m)$ such that

(a) $T \vdash_{WT} a_i :: \tau_i$ for all $(1 \leq i \leq m)$ and $T \vdash_{WT} h :: \overline{\tau}_m \to \tau$.

(b) $T \vdash_{WT} b_i :: \tau_i'$ for all $(1 \leq i \leq m)$ and $T \vdash_{WT} h :: \overline{\tau'}_m \to \tau$.

Consider the type environments $T_1 = \{X_1 :: \tau_1, \ldots, X_m :: \tau_m\}$ and $T_2 = \{X_1 :: \tau_1', \ldots, X_m :: \tau_m'\}$. Due to (a) and (b), we obtain $T_1 \vdash_{WT} h\,\overline{X}_m :: \tau$ and $T_2 \vdash_{WT} h\,\overline{X}_m :: \tau$. By applying Lemma 6 to T_1, T_2 and the transparent pattern $h\,\overline{X}_m$, we can conclude that $\tau_i = \tau_i'$ for all $(1 \leq i \leq m)$, which completes the proof. □

6.2 Proofs of Results from Subsection 3.1

Proof of Lemma 9: Since the given defining rule is well-typed, we can assume: (a) $T_0 \vdash_{WT} t_i :: \tau_i$ for all $1 \leq i \leq n$, (b) $T_0 \vdash_{WT} r :: \tau_0$. By the assumptions of the lemma, we also have (c) $T \vdash_{WT} t_i\sigma_d :: \tau_i\sigma_t$ for all $1 \leq i \leq n$. From (a) and the Type Instantiation Lemma 4, we get (d) $T_0\sigma_t \vdash_{WT} t_i :: \tau_i\sigma_t$ for all $1 \leq i \leq n$. On the other hand, taking into account (c) and the linearity of $f\,\overline{t}_n$ we can build a type environment T_0' with $dom(T_0') = var(f\,\overline{t}_n)$ and such that (e) $T_0' \vdash_{WT} t_i :: \tau_i\sigma_t$ for all $1 \leq i \leq n$, (f) $T \vdash_{WT} X\sigma_d :: T_0'(X)$ for all $X \in var(f\,\overline{t}_n)$. To build T_0', it

is enough to choose as $T_0'(X)$ the type τ_X inferred for $X\sigma_d$ as part of the type inference ensured by (c) (for that t_i in which X occurs). Note that $t_1, \ldots, t_n$ are transparent patterns without variables in common. Therefore we can apply the Transparency Lemma 6 to (d), (e) and we obtain: (g) $T_0'(X) = (T_0\sigma_t)(X) = T_0(X)\sigma_t$ for all $X \in var(f\ \overline{t}_n)$ From (g), (f) and $var(r) \subseteq var(f\ \overline{t}_n)$, we get: (h) $T \vdash_{WT} X\sigma_d :: T_0(X)\sigma_t$ for all $X \in var(r)$. Finally, (b) and (h) allow us to apply the Well-typed Substitution Lemma 3, which gives $T \vdash_{WT} r\sigma_d :: \tau\sigma_t$, as we wanted to prove. □

Proof of Theorem 2: Assume a given GORC proof Π for $\mathcal{P} \vdash_{GORC} e \to t$, with size $k \geq 1$, measured as the number of GORC inference steps. We reason by induction over k.

In the *base case*, $k = 1$ and there are three possible cases:

Case **BT***:* In this case Π consists of one single application of **BT**. Then $t = \bot$, and $T \vdash_{WT} \bot :: \tau$ follows from the type inference rule **ID**, since $(\bot :: \alpha) \in_{var} \Sigma_\bot$.

Case **RR***:* If Π consists of one single application of **RR**, then $e = t = X$, for some variable X, and the conclusion is trivial.

Case **DC***:* In this case Π must consist of one single application of **DC** with $m = 0$, since $m > 0$ would yield size greater that 1 for Π. Therefore, $e = t = h \in DC \cup FS$, and the conclusion is trivial again.

For the *induction step*, we assume $k > 1$ and we analyze the two possible cases according to the GORC inference rule applied at the last step of Π:

Case **DC***:* In this case, $e \to t$ has the form of the conclusion of the GORC-rule **DC**, with $m \geq 1$. The assumption $T \vdash_{WT} e :: \tau$ becomes $T \vdash_{WT} h\overline{e}_m :: \tau$, which implies the existence of types τ_i such that (a) $T \vdash_{WT} e_i :: \tau_i$ for all $1 \leq i \leq m$, (b) $T \vdash_{WT} h :: \overline{\tau}_m \to \tau$. For each $1 \leq i \leq m$, we have $\mathcal{P} \vdash_{GORC} e_i \to t_i$ with a GORC proof of size less than k. By induction hypothesis, we can assume: (c) $T \vdash_{WT} t_i :: \tau_i$ for all $1 \leq i \leq m$. From (b) and (c), we can conclude $T \vdash_{WT} h\,\overline{t}_m :: \tau$, which is the same as $T \vdash_{WT} t :: \tau$.

Case **OR***:* Now we can assume that $e \to t$ has the form of the conclusion of the GORC-rule **OR**, where each premise has a GORC proof of size smaller than k. In particular, $e = f\ \overline{e}_n\ \overline{a}_m$ for some $f \in FS^n$, and the assumption $T \vdash_{WT} e :: \tau$, implies the existence of types τ_i, μ_j such that: (d) $T \vdash_{WT} e_i :: \tau_i$ for all $1 \leq i \leq n$, (e) $T \vdash_{WT} a_j :: \mu_j$ for all $1 \leq j \leq m$, (f) $T \vdash_{WT} f :: \overline{\tau}_n \to$

$\overline{\mu}_m \to \tau$. By the induction hypothesis applied to $e_i \to t_i$, we can assume: (g) $T \vdash_{WT} t_i :: \tau_i$ for all $1 \leq i \leq n$. Due to (f) and (g), we can apply Lemma 9 to $(f\,\overline{t}_n \to r \Leftarrow C) \in [\mathcal{P}]_\bot$, and we obtain (h) $T \vdash_{WT} r :: \overline{\mu}_m \to \tau$. From (e) and (h) we can infer $T \vdash_{WT} r\,\overline{a}_m :: \tau$. Finally, we can apply the induction hypothesis to $r\,\overline{a}_m \to t$ to conclude $T \vdash_{WT} t :: \tau$. Note that our reasoning in this case ignores the GORC proofs for C in the premises of **OR**. In fact, some of these conditions could be ill-typed. This eventuality does not contradict the Subject Reduction Theorem. □

6.3 Proofs of Results from Subsection 3.2

Proof of Proposition 2: The four items can be proved straightforwardly, using structural induction over τ, e or t, according to the case. A similar result for FO CRWL can be found in [18] as Proposition 5.1. Here we limit ourselves to give the proof of item 3, reasoning by induction of the structure of a partial pattern $t \in Pat_\bot$. There are four cases to consider:

Case $t = X$, $X \in DVar$: This is a base case. $[\![X]\!]^{\mathcal{A}}\eta_d = \langle \eta_d(X) \rangle$, where $\eta_d(X) \in DefVal(\mathcal{A})$, if $\eta \in DefVal(\mathcal{A})$.

Case $t = \bot$: This is also a base case. $[\![\bot]\!]^{\mathcal{A}}\eta_d = \langle \bot \rangle$.

Case $t = c\,\overline{t}_m$, $c \in DC^n$, $0 \leq m \leq n$: By induction hypothesis, we can assume elements $v_i \in D^{\mathcal{A}}$ such that $[\![t_i]\!]^{\mathcal{A}}\eta_d = \langle v_i \rangle$ $(1 \leq i \leq m)$. In case that $\eta \in DefVal(\mathcal{A})$ and the t_i are total patterns, we can assume $v_i \in Def(D^{\mathcal{A}})$. By item 9 of Definition 5 and monotonicity of $@^{\mathcal{A}}$, we get

$$[\![c\,\overline{t}_m]\!]^{\mathcal{A}}\eta_d = c^{\mathcal{A}}\,@^{\mathcal{A}}\,v_1\,@^{\mathcal{A}}\,\ldots\,@^{\mathcal{A}}\,v_m = \langle v \rangle$$

for some $v \in D^{\mathcal{A}}$, and $v \in Def(D^{\mathcal{A}})$ in the case that $v_1, \ldots, v_m \in Def(D^{\mathcal{A}})$.

Case $t = f\,\overline{t}_m$, $f \in FS^n$, $0 \leq m < n$: This case is similar to the previous case, also using the induction hypothesis and item 9 of Definition 5. □

Proof of Proposition 3: We reason by induction on the size n (measured as the number of inference steps) of a given derivation of the type judgement $T \vdash_{WT} e :: \tau$.

In the *base case*, $n = 1$ and there are two subcases:

Case **VT***:* In this case e is a variable X and $T \vdash_{WT} X :: \tau$ is proved by one single **VR** step. Then $[\![X]\!]^{\mathcal{A}}\eta_d = \langle \eta_d(X) \rangle \subseteq \mathcal{E}^{\mathcal{A}}([\![\tau]\!]^{\mathcal{A}}\eta_t)$, since η is well-typed w.r.t. T.

Case **ID***:* In this case $e = h \in DC \cup FS$ with principal type declaration $(h :: \tau_0) \in_{var} \Sigma_\perp$. Note that $h = \perp$ is possible here. Necessarily, $T \vdash_{WT} h :: \tau$ has been proved by one single **ID** step, and $\tau = \tau_0\sigma_t$ for some $\sigma_t \in TSub$. By the Substitution Lemma 10, $[\![\tau]\!]^{\mathcal{A}}\eta_t = [\![\tau_0]\!]^{\mathcal{A}}\eta\sigma_t$. On the other hand, $[\![h]\!]^{\mathcal{A}}\eta_d$ equals $h^{\mathcal{A}}$ if $h \in FS^0$ and $\langle h^{\mathcal{A}}\rangle$ otherwise. In both cases, $[\![h]\!]^{\mathcal{A}}\eta_d \subseteq \mathcal{E}^{\mathcal{A}}([\![\tau_0]\!]^{\mathcal{A}}\eta\sigma_t)$ because $\mathcal{A}$ is a well-typed algebra.

For the *induction step*, we consider a derivation of $T \vdash_{WT} (e\ e_1) :: \tau$ in n steps, $n > 1$. The last step must use **AP**, and there must be shorter derivations proving $T \vdash_{WT} e :: \tau_1 \to \tau$ and $T \vdash_{WT} e_1 :: \tau_1$ for some type τ_1. By induction hypothesis, we can assume: (a) $[\![e]\!]^{\mathcal{A}}\eta_d \subseteq \mathcal{E}^{\mathcal{A}}([\![\tau_1 \to \tau]\!]^{\mathcal{A}}\eta_t)$ and $[\![e_1]\!]^{\mathcal{A}}\eta_d \subseteq \mathcal{E}^{\mathcal{A}}([\![\tau_1]\!]^{\mathcal{A}}\eta_t)$. Since $\mathcal{A}$ is a well-typed algebra, we know that: (b) $u\ @^{\mathcal{A}}\ u_1 \subseteq \mathcal{E}^{\mathcal{A}}([\![\tau]\!]^{\mathcal{A}}\eta_t)$ for all $u \in \mathcal{E}^{\mathcal{A}}([\![\tau_1 \to \tau]\!]^{\mathcal{A}}\eta_t)$, $u_1 \in \mathcal{E}^{\mathcal{A}}([\![\tau_1]\!]^{\mathcal{A}}\eta_t)$. On the other hand, we also have: (c) $[\![(e\ e_1)]\!]^{\mathcal{A}}\eta_d = [\![e]\!]^{\mathcal{A}}\eta_d\ @^{\mathcal{A}}\ [\![e_1]\!]^{\mathcal{A}}\eta_d = \bigcup\{u\ @^{\mathcal{A}}\ u_1 \mid u \in [\![e]\!]^{\mathcal{A}}\eta_d, u_1 \in [\![e_1]\!]^{\mathcal{A}}\eta_d\}$. From (a), (b), (c) it follows that $[\![(e\ e_1)]\!]^{\mathcal{A}}\eta_d \subseteq \mathcal{E}^{\mathcal{A}}([\![\tau]\!]^{\mathcal{A}}\eta_t)$, as we wanted to prove. □

Proof of Lemma 11: To prove item 1 we have to check that all the technical conditions stated in the Definitions 5, 6 are fulfilled. The crucial points are the following ones:

1(a) $@^{\mathcal{M}_{\mathcal{P}}(T)}$ must be monotonic mapping returning cones as values. This follows from items 2 and 3 of Lemma 8. The key fact is that, for patterns $t_i, t'_i \in Pat_\perp$ such that $t_i \sqsupseteq t'_i$, one gets:

$$\{t \in Pat_\perp \mid \mathcal{P} \vdash_{GORC} f\ \overline{t}_m \to t\} \subseteq \{t \in Pat_\perp \mid \mathcal{P} \vdash_{GORC} f\ \overline{t'}_m \to t\}$$

where both sets are cones.

1(b) For every $\tau \in Type$, the set $\mathcal{E}^{\mathcal{M}_{\mathcal{P}}(T)}(\tau) = \{t \in Pat_\perp \mid T \vdash_{WT} t :: \tau\}$ must be a cone. Indeed, $T \vdash_{WT} \perp :: \tau$ holds because of the type inferring rule **ID**. Moreover, assuming $T \vdash_{WT} t' :: \tau$ and $t \sqsupseteq t'$, $T \vdash_{WT} t :: \tau$ follows from Lemma 2.

1(c) For every $f \in FS^0$, the set $f^{\mathcal{M}_{\mathcal{P}}(T)} = \{t \in Pat_\perp \mid \mathcal{P} \vdash_{GORC} f \to t\}$ must be a cone. This is true because of item 2 from Lemma 8.

To prove item 2, let us assume that $\mathcal{P}$ is a well-typed program. Considering item 4 of the Proposition 2, as well as the construction of $\mathcal{M}_{\mathcal{P}}(T)$, proving that $\mathcal{M}_{\mathcal{P}}(T)$ is well-typed reduces to check the following conditions:

2(a) $T \vdash_{WT} c :: \tau\sigma_t$, for every $c :: \tau \in DC^n$, $\sigma_t \in TSub$.

2(b) $T \vdash_{WT} f :: \tau\sigma_t$, for every $f :: \tau \in FS^n$, $\sigma_t \in TSub$.

2(c) For every $(f :: \overline{\tau}_n \to \tau_0) \in FS^n$, $n \geq 0$, for every $t_i, t_i' \in Pat_\perp$, for every $\sigma_t \in TSub$: $T \vdash_{WT} t_i :: \tau_i\sigma_t$ $(1 \leq i \leq n)$ and $\mathcal{P} \vdash_{GORC} f\,\overline{t}_n \to t \implies T \vdash_{WT} t :: \tau_0\sigma_t$.

Indeed, 2(a) and 2(b) follows from the type inference rule **ID**. On the other hand, the assumptions in 2(c) imply $T \vdash_{WT} f\,\overline{t}_n :: \tau_0\sigma_t$, which allows to deduce $T \vdash_{WT} t :: \tau_0\sigma_t$ by an application of the Subject Reduction Theorem 2. □

Proof of Lemma 12: We start by showing that item 2 is an easy consequence of item 1. The statement φ can be either $e \to t$ or $a \bowtie b$. In the first case, we have:

$$(\mathcal{M}_\mathcal{P}(T), \sigma_d) \vDash e \to t \iff [\![t]\!]^{\mathcal{M}_\mathcal{P}(T)}\sigma_d \subseteq [\![e]\!]^{\mathcal{M}_\mathcal{P}(T)}\sigma_d \iff$$
$$t\sigma_d \in [\![e]\!]^{\mathcal{M}_\mathcal{P}(T)}\sigma_d \iff \mathcal{P} \vdash_{GORC} e\sigma_d \to t\sigma_d$$

where the first equivalence is justified by Definition 8, the second equivalence is true because of item 4 in Proposition 2 and the third equivalence holds by item 1 of the present lemma.

In the case $\varphi = a \bowtie b$ we reason as follows: $(\mathcal{M}_\mathcal{P}(T), \sigma_d) \vDash a \bowtie b \iff$ there is a total pattern $t \in [\![a]\!]^{\mathcal{M}_\mathcal{P}(T)}\sigma_d \cap [\![b]\!]^{\mathcal{M}_\mathcal{P}(T)}\sigma_d \iff$ there is a pattern $t \in Pat$ such that $\mathcal{P} \vdash_{GORC} a\sigma_d \to t$ and $\mathcal{P} \vdash_{GORC} b\sigma_d \to t \iff \mathcal{P} \vdash_{GORC} a\sigma_d \bowtie b\sigma_d$, where the first equivalence is justified by Definition 8, the second equivalence is true by item 1 of this lemma, and the third equivalence is justified by the GORC rule **JN**.

In order to prove item 1, we observe that $(*)$ $[\![e\sigma_d]\!]^{\mathcal{M}_\mathcal{P}(T)}id = [\![e]\!]^{\mathcal{M}_\mathcal{P}(T)}id\sigma_d = [\![e]\!]^{\mathcal{M}_\mathcal{P}(T)}\sigma_d$, where the first identity holds because of the Substitution Lemma 10 and the second identity is true because $[\![\sigma_d(X)]\!]^{\mathcal{M}_\mathcal{P}(T)}id = \langle\sigma_d(X)\rangle$. We will show that every $e \in Exp_\perp$ verifies $(**)$ $[\![e]\!]^{\mathcal{M}_\mathcal{P}(T)}id = \{t \in Pat_\perp \mid \mathcal{P} \vdash_{GORC} e \to t\}$ Note that item 1 follows from $(*)$, $(**)$ (with e replaced by $e\sigma_d$ in $(**)$). To simplify the notation in the rest of this proof, we abbreviate "$[\![e]\!]^{\mathcal{M}_\mathcal{P}(T)}id$" as "$[\![e]\!]^{\mathcal{P}}$" and "$\mathcal{P} \vdash_{GORC} e \to t$" as "$e \to_\mathcal{P} t$". We give separate proofs for the two inclusions: (A) $[\![e]\!]^{\mathcal{P}} \subseteq \{t \in Pat_\perp \mid e \to_\mathcal{P} t\}$ and (B) $\{t \in Pat_\perp \mid e \to_\mathcal{P} t\} \subseteq [\![e]\!]^{\mathcal{P}}$.

Proof of (A) Due to the construction of $\mathcal{M}_\mathcal{P}(T)$ and the recursive definition of expression evaluation, the true facts of the form "$t \in [\![e]\!]^{\mathcal{P}}$" are exactly

those that can be derived in finitely many steps by means of the following rules:

(D1) $\bot \in [\![e]\!]^{\mathcal{P}}$, for all $e \in Exp_{\bot}$.

(D2) $X \in [\![X]\!]^{\mathcal{P}}$, for all $X \in DVar$.

(D3) $t_i \in [\![e_i]\!]^{\mathcal{P}}$ $(1 \leq i \leq m) \implies h\,\overline{t}_m \in [\![h\,\overline{e}_m]\!]^{\mathcal{P}}$, for all rigid and passive expressions $h\,\overline{e}_m$.

(D4) $t_i \in [\![e_i]\!]^{\mathcal{P}}$ $(1 \leq i \leq n)$, $u \in [\![f\,\overline{t}_n]\!]^{\mathcal{P}}$, $t \in [\![u\,\overline{a}_m]\!]^{\mathcal{P}} \implies t \in [\![f\,\overline{e}_n\,\overline{a}_m]\!]^{\mathcal{P}}$, for all rigid and active expressions $f\,\overline{e}_n\,\overline{a}_m$.

Assume that "$t \in [\![e]\!]^{\mathcal{P}}$" has been derived in k steps by means of the rules (D1)–(D4) above. We use induction over k to show that $e \to_{\mathcal{P}} t$.

In the *base case* $k = 1$, $t \in [\![e]\!]^{\mathcal{P}}$ has been derived by one single application of (D1), (D2) or (D3) with $m = 0$. In all these cases, $e \to_{\mathcal{P}} t$ is trivial.

For the *inductive case*, $k > 1$, the last step in the derivation of $t \in [\![e]\!]^{\mathcal{P}}$ must correspond to rule (D3) with $m > 0$, or to rule (D4). In the (D3) subcase, we get $e = h\,\overline{e}_m$ and $t = h\,\overline{t}_m$ where $e_i \to_{\mathcal{P}} t_i$ $(1 \leq i \leq m)$ can be assumed by induction hypothesis. Therefore, $e \to_{\mathcal{P}} t$ follows by the GORC rule **DC**. In the (D4) subcase $e = f\,\overline{e}_n\,\overline{a}_m$, and by induction hypothesis we obtain $e_i \to_{\mathcal{P}} t_i$ $(1 \leq i \leq n)$ and $u\,\overline{a}_m \to_{\mathcal{P}} t$, where $u \in [\![f\,\overline{t}_n]\!]^{\mathcal{P}}$. By construction of $\mathcal{M}_{\mathcal{P}}(T)$, $u \in [\![f\,\overline{t}_n]\!]^{\mathcal{P}}$ implies $f\,\overline{t}_n \to_{\mathcal{P}} u$. By Proposition 1, GORC is equivalent to BRC, where reduction is transitive and preserved by contexts. Therefore, we can combine the reductions $e_i \to_{\mathcal{P}} t_i$ $(1 \leq i \leq n)$; $f\,\overline{t}_n \to_{\mathcal{P}} u$; $u\,\overline{a}_m \to_{\mathcal{P}} t$ to obtain:

$$f\,\overline{e}_n\,\overline{a}_m \to_{\mathcal{P}} f\,\overline{t}_n\,\overline{a}_m \to_{\mathcal{P}} u\,\overline{a}_m \to_{\mathcal{P}} t$$

which proves $f\,\overline{e}_n\,\overline{a}_m \to_{\mathcal{P}} t$, as desired

Proof of (B) We assume that $e \to_{\mathcal{P}} t$ has been established be means of a GORC proof of size k, and we use induction over k to show $t \in [\![e]\!]^{\mathcal{P}}$.

The *base case*, $k = 1$, corresponds to a GORC proof consisting of one single application of **BT**, **RR** or **DC** with $m = 0$. In all these situations, $t \in [\![e]\!]^{\mathcal{P}}$ is obvious.

For the *inductive case*, $k > 1$, we distinguish two subcases, according to the inference rule used at the last step of the given GORC proof.

Case **DC** *with* $m > 1$*:* We have $e = h\,\overline{e}_m$, $t = h\,\overline{t}_m$, and $t_i \in [\![e_i]\!]^{\mathcal{P}}$ $(1 \leq i \leq m)$ can be assumed by induction hypothesis. Then $h\,\overline{t}_m \in [\![h\,\overline{e}_m)]\!]^{\mathcal{P}}$ follows by construction of $\mathcal{M}_{\mathcal{P}}(T)$.

Case **OR***:* We have a rule instance $(f\,\overline{t}_n \to r \Leftarrow C) \in [\mathcal{P}]_{\perp}$ such that (a) $e_i \to_{\mathcal{P}} t_i$ $(1 \leq i \leq n)$, (b) $\mathcal{P} \vdash_{GORC} C$, (c) $r\,\overline{a}_m \to_{\mathcal{P}} t$. By induction hypothesis applied to (a), (c), we get $t_i \in [\![e_i]\!]^{\mathcal{P}}$ $(1 \leq i \leq n)$ and $t \in [\![r\,\overline{a}_m]\!]^{\mathcal{P}}$. These facts, together with the monotonicity of the apply operation in any algebra and the definition of expression evaluation, imply the following: (d) $t \in [\![r\,\overline{a}_m]\!]^{\mathcal{P}}$ and $[\![f\,\overline{t}_n\,\overline{a}_m]\!]^{\mathcal{P}} \subseteq [\![f\,\overline{e}_n\,\overline{a}_m]\!]^{\mathcal{P}}$. By (d), it is enough to show that $[\![r\,\overline{a}_m]\!]^{\mathcal{P}} \subseteq [\![f\,\overline{t}_n\,\overline{a}_m]\!]^{\mathcal{P}}$, which in turn follows from $[\![r]\!]^{\mathcal{P}} \subseteq [\![f\,\overline{t}_n]\!]^{\mathcal{P}}$. To show this, let us assume $u \in [\![r]\!]^{\mathcal{P}}$. Because of the inclusion (A) (already proved above) we obtain: (e) $r \to_{\mathcal{P}} u$. Moreover, due to item 1 from Lemma 8, we also have (f) $t_i \to_{\mathcal{P}} t_i$ $(1 \leq i \leq n)$. Now, $f\,\overline{t}_n \to_{\mathcal{P}} u$ follows from (f), (b), (e) and the GORC rule **OR**. Due to the construction of $\mathcal{M}_{\mathcal{P}}(T)$, $f\,\overline{t}_n \to_{\mathcal{P}} u$ implies $u \in [\![f\,\overline{t}_n]\!]^{\mathcal{P}}$, which completes the proof of $[\![r]\!]^{\mathcal{P}} \subseteq [\![f\,\overline{t}_n]\!]^{\mathcal{P}}$ and the Lemma. □

Proof of Theorem 3: In order to prove $\mathcal{M}_{\mathcal{P}}(T) \models \mathcal{P}$, we consider any defining rule $(f\,\overline{t}_n \to r \Leftarrow C \;\square\; T) \in \mathcal{P}$ and any data substitution $\sigma_d \in DSub_{\perp}$ (which is the same as a data valuation over $\mathcal{M}_{\mathcal{P}}(T)$). Assume that $(\mathcal{M}_{\mathcal{P}}(T), \sigma_d) \models C$. By item 2 from Lemma 12, this means that $\mathcal{P} \vdash_{GORC} C\sigma_d$. We have to check $[\![r]\!]^{\mathcal{M}_{\mathcal{P}}(T)}\sigma_d \subseteq [\![f\,\overline{t}_n]\!]^{\mathcal{M}_{\mathcal{P}}(T)}\sigma_d$. By item 1 from Lemma 12, this amounts to show:

$$\{t \in Pat_{\perp} \mid \mathcal{P} \vdash_{GORC} r\sigma_d \to t\} \subseteq \{t \in Pat_{\perp} \mid \mathcal{P} \vdash_{GORC} (f\,\overline{t}_n)\sigma_d \to t\}$$

This is true, because for any $t \in Pat_{\perp}$ such that $\mathcal{P} \vdash_{GORC} r\sigma_d \to t$, we can derive $\mathcal{P} \vdash_{GORC} (f\,\overline{t}_n)\sigma_d \to t$ in one **OR** step, using $\mathcal{P} \vdash_{GORC} C\sigma_d$ (which we are assuming) and $\mathcal{P} \vdash_{GORC} t_i\sigma_d \to t_i\sigma_d$ (which is true by item 1 from Lemma 8). Next, we show the equivalence of the items in the Theorem's statement by proving three implications.

1 $\Longrightarrow$ 2: Let us fix any model $\mathcal{A} \models \mathcal{P}$ and any totally defined valuation $\eta \in DefVal(\mathcal{A})$. We assume $\mathcal{P} \vdash_{GORC} \varphi$ with some proof Π of size k, and we reason by induction over k to prove $(\mathcal{A}, \eta) \models \varphi$.

In the *base case*, $k = 1$, φ must be either $e \to \perp$ or $X \to X$ or $h \to h$, and $(\mathcal{A}, \eta) \models \varphi$ is trivially true.

In the *inductive case*, $k > 1$, we must consider three subcases.

(a) The proof Π ends with a **DC** step. Then $\varphi = h\,\overline{e}_m \to h\,\overline{t}_m$, and by induction hypothesis applied to the premises of the **DC** inference, we get $(\mathcal{A}, \eta) \models e_i \to t_i$, which amounts to $[\![t_i]\!]^{\mathcal{A}}\eta \subseteq [\![e_i]\!]^{\mathcal{A}}\eta$, for all $1 \leq i \leq n$, $[\![h\,\overline{t}_m]\!]^{\mathcal{A}}\eta \subseteq [\![h\,\overline{e}_m]\!]^{\mathcal{A}}\eta$ is easy to check, using the definition of expression evaluation. Therefore, $(\mathcal{A}, \eta) \models h\,\overline{e}_m \to h\,\overline{t}_m$.

(b) The proof Π ends with an **OR** step. Then $\varphi = f\,\overline{e}_n\,\overline{a}_m \to t$, and by induction hypothesis applied to the premises of the **OR** inference, we get (b1) $(\mathcal{A}, \eta) \models e_i \to t_i$, i.e. $[\![t_i]\!]^{\mathcal{A}}\eta \subseteq [\![e_i]\!]^{\mathcal{A}}\eta$, $(1 \leq i \leq n)$, (b2) $(\mathcal{A}, \eta) \models C$, (b3) $(\mathcal{A}, \eta) \models r\,\overline{a}_m \to t$, i.e. $[\![t]\!]^{\mathcal{A}}\eta \subseteq [\![r\,\overline{a}_m]\!]^{\mathcal{A}}\eta$ where $(f\,\overline{t}_n \to r \Leftarrow C) \in [\mathcal{P}]_\perp$ is the instance of $\mathcal{P}$-rule used in the **OR** step. Since $\mathcal{A}$ is a model of $\mathcal{P}$, $\mathcal{A}$ satisfies all the rewrite rules in $\mathcal{P}$. Due to Lemma 10, $\mathcal{A}$ satisfies also all the instances of rules in $\mathcal{P}$, in particular, $(f\,\overline{t}_n \to r \Leftarrow C)$. Then, from (b2) we can infer (b4) $(\mathcal{A}, \eta) \models f\,\overline{t}_n \to r$, i.e. $[\![r]\!]^{\mathcal{A}}\eta \subseteq [\![f\,\overline{t}_n]\!]^{\mathcal{A}}\eta$. Considering the definition of expression evaluation, it is easy to see that (b3), (b4) and (b1) can be applied successively, leading to the inclusions:

$$[\![t]\!]^{\mathcal{A}}\eta \subseteq [\![r\,\overline{a}_m]\!]^{\mathcal{A}}\eta \subseteq [\![f\,\overline{t}_n\,\overline{a}_m]\!]^{\mathcal{A}}\eta \subseteq [\![f\,\overline{e}_n\,\overline{a}_m]\!]^{\mathcal{A}}\eta$$

This entails $[\![t]\!]^{\mathcal{A}}\eta \subseteq [\![f\,\overline{e}_n\,\overline{a}_m]\!]^{\mathcal{A}}\eta$, i.e. $(\mathcal{A}, \eta) \models f\,\overline{e}_n\,\overline{a}_m \to t$.

(c) The proof Π ends with a **JN** step. Then $\varphi = a \bowtie b$, and by induction hypothesis applied to the premises of the **JN** inference, we get a total pattern $t \in Pat$ such that $[\![t]\!]^{\mathcal{A}}\eta \subseteq [\![a]\!]^{\mathcal{A}}\eta$ and $[\![t]\!]^{\mathcal{A}}\eta \subseteq [\![b]\!]^{\mathcal{A}}\eta$. Since η is totally defined, we can assume $[\![t]\!]^{\mathcal{A}}\eta = \langle v \rangle$, for some $v \in Def(D^{\mathcal{A}})$, by item 3 of Proposition 2. Then there exists a totally defined element $v \in ([\![a]\!]^{\mathcal{A}}\eta \cap [\![b]\!]^{\mathcal{A}}\eta)$, which means $(\mathcal{A}, \eta) \models a \bowtie b$.

2 $\Longrightarrow$ 3: This implication holds because $\mathcal{M}_{\mathcal{P}}(T) \models \mathcal{P}$, as we have proved already, and id is a totally defined valuation.

3 $\Longrightarrow$ 1: This follows from item 2 of Lemma 12, taking the identity substitution id as σ_d. □

6.4 Proofs of Results from Subsection 4.1

Proof of Lemma 13:
(a) $\Longrightarrow$ (b): Assume that G is well-typed, and take $T' = \hat{T}$. By the Type Reconstruction Theorem 1, $\hat{T} \vdash_{WT} (P, C, S_d) :: \texttt{bool}$.

(b) $\Longrightarrow$ (c): This follows from the form of the type inference rules and the principal types of the "operations" $(\to)$, $(\bowtie)$, $(\approx)$ and $(,)$.

(c) $\Longrightarrow$ (a): Assume $T \leq T'$ for which *(c1)*, *(c2)* and *(c3)* holds. Then it is easy to see that $T' \vdash_{WT} (P, C, S_d) :: \texttt{bool}$. Then, the Type Reconstruction Theorem 1 ensures $TR(T, (P, C, S_d)) = ((-)^\tau, E))$ and $\sigma_t \in TSol(E)$ such that $T\sigma_t = T'$ and $\tau\sigma_t = \texttt{bool}$. In particular, G is well-typed because $TSol(E) \neq \emptyset$. □

Proof of Lemma 14:
Item 1: To show the inclusion, let us assume $(R, \theta) \in WTSol(\hat{G})$. Then $\theta_d \in Sol(G)$ and also: (a) $dom(R) \subseteq ran(\theta_d) \setminus dom(\hat{T})$, (b) $\theta_t \in TSol(\hat{S}_t)$, (c) $(\hat{T}\theta_t \cup R) \vdash_{WT} X\theta_d :: \hat{T}(X)\theta_t$, for all $X \in dom(\hat{T})$. To show $(R, \theta) \in WTSol(G)$ we need: (a′) $dom(R) \subseteq ran(\theta_d) \setminus dom(T)$, (b′) $\theta_t \in TSol(S_t)$, (c′) $(T\theta_t \cup R) \vdash_{WT} X\theta_d :: T(X)\theta_t$, for all $X \in dom(T)$. By construction of $\hat{G}$ from G, we have $\hat{T} = T\hat{\sigma}_t$, where $\hat{\sigma}_t = mgu(S_t, E)$. Then (a) $\Longleftrightarrow$ (a′) and (b) $\Longrightarrow$ (b′). Moreover, (b) also implies $\theta_t = \hat{\sigma}_t\theta_t$, which entails $\hat{T}\theta_t = T\hat{\sigma}_t\theta_t = T\theta_t$ and also $\hat{T}(X)\theta_t = T(X)\hat{\sigma}_t\theta_t = T(X)\theta_t$ for all $X \in dom(T) = dom(\hat{T})$. This being the case, (c) $\Longleftrightarrow$ (c′).

To show that the opposite inclusion can fail, consider $G \neq \hat{G}$ given as follows:

$$G = \emptyset \ \square\ \texttt{[X|Xs]} \bowtie \texttt{[Y|Ys]} \ \square\ \emptyset \ \square\ \emptyset \ \square\ \{\texttt{X} :: \alpha,\ \texttt{Xs} :: [\beta],\ \texttt{Y} :: \alpha,\ \texttt{Ys} :: [\beta]\}$$

$$\hat{G} = \emptyset \ \square\ \texttt{[X|Xs]} \bowtie \texttt{[Y|Ys]} \ \square\ \emptyset \ \square\ \beta \approx \alpha \ \square\ \{\texttt{X}, \texttt{Y} :: \alpha;\ \texttt{Xs}, \texttt{Ys} :: [\alpha]\}$$

Then, with the substitutions $\theta_t = \{\alpha \mapsto \texttt{nat}, \beta \mapsto \texttt{bool}\}$ and $\theta_d = \{\texttt{X} \mapsto \texttt{z}, \texttt{Xs} \mapsto \texttt{[true]}, \texttt{Y} \mapsto \texttt{z}, \texttt{Ys} \mapsto \texttt{[true]}\}$ we get $(\emptyset, \theta) \in WTSol(G) \setminus WTSol(\hat{G})$.

Item 2: Assume $(a \bowtie b) \in C$. Since G is well-typed and $\hat{G} = G$, there must be some $\tau \in Type$ such that (a) $T \vdash_{WT} a :: \tau$ and $T \vdash_{WT} b :: \tau$. On the other hand, since $(R, \theta) \in WTSol(G)$, we have (b) $(T\theta_t \cup R) \vdash_{WT} X\theta_d :: T(X)\theta_t$, for all $X \in var(a) \cup var(b)$. Now, (a), (b) match the hypothesis of Lemma 3 (with T as T_0 and $T\theta_t \cup R$ as T_1) and we can conclude $(T\theta_t \cup R) \vdash_{WT} a\theta_d :: \tau\theta_t$ and $(T\theta_t \cup R) \vdash_{WT} b\theta_d :: \tau\theta_t$. An analogous reasoning can be made for $(e \rightarrow t) \in P$ and $(X \approx t) \in S_d$.

To show that the result can fail if $G \neq \hat{G}$, consider G, $\hat{G}$ and θ as in the previous item. Then $(\emptyset, \theta) \in WTSol(G)$, but $T\theta_t = \{\texttt{X}, \texttt{Y} :: \texttt{nat}; \texttt{Xs}, \texttt{Ys} :: \texttt{[bool]}\} \not\vdash_{WT} (\texttt{[X|Xs]} \bowtie \texttt{[Y|Ys]})\theta_d = \texttt{[z|true]} \bowtie \texttt{[z|true]}$. □

6.5 Proofs of Results from Subsection 4.2

Proof of Theorem 4: We rely on some basic facts from computability theory, which can be found in textbooks such as [8]. In particular, we recall the following well-known, undecidable problem K: given a natural number n, tell if $\varphi_n(n)\downarrow$ holds. Here, the notation $\varphi_n(n)\downarrow$ means that the computable function φ_n with index n eventually halts when given n itself as argument.

Using the type `nat` and the constructors `z`, `s` from Example 2, it is possible to write a well-typed simple program $\mathcal{P}$ including defining rules for a function $\texttt{halt} :: \texttt{nat} \to \texttt{nat}$ such that, for all $n \in \mathbb{N}$, one has:

- If $\varphi_n(n)$ halts in k steps, the evaluation of $\texttt{halt}\ (\texttt{s}^n\ \texttt{z})$ w.r.t. $\mathcal{P}$ computes the result $\texttt{s}^k\ \texttt{z}$.
- If $\varphi_n(n)$ does not halt, the evaluation of $\texttt{halt}\ (\texttt{s}^n\ \texttt{z})$ w.r.t. $\mathcal{P}$ does not terminate, but computes the potentially infinite result $\texttt{s(s(s(s}\ldots$.

In the two items above, the notation $\texttt{s}^n\ \texttt{z}$ refers to the natural representation of $n \in \mathbb{N}$ using the data constructors `z` and `s`. Building the program $\mathcal{P}$ so that `halt` behaves as described is possible, because simple CRWL programs are powerful enough to simulate Turing machines (or any other Turing-complete computation model). We omit the tedious proof of this claim, that would follow well-known techniques from computability theory.

To continue with our proof, let us assume that program $\mathcal{P}$ is expanded to include the defining rules for the functions `tail` and `second`, as given in Example 2, as well as the following defining rules for four new functions:

$$\texttt{oh, fefo} :: \texttt{nat} \to (\alpha \to \alpha)$$
$$\texttt{oh N} \to \texttt{fefo (halt N)} \Leftarrow \emptyset \ \square\ \{\texttt{N} :: \texttt{nat}\}$$
$$\texttt{fefo z} \to \texttt{snd (tail [true])} \Leftarrow \emptyset \ \square\ \emptyset$$
$$\texttt{fefo (s N)} \to \texttt{fefo N} \Leftarrow \emptyset \ \square\ \{\texttt{N} :: \texttt{nat}\}$$

$$\texttt{ah, fefa} :: \texttt{nat} \to (\alpha \to \alpha)$$
$$\texttt{ah N} \to \texttt{fefa (halt N)} \Leftarrow \emptyset \ \square\ \{\texttt{N} :: \texttt{nat}\}$$
$$\texttt{fefa z} \to \texttt{snd (tail [z])} \Leftarrow \emptyset \ \square\ \emptyset$$
$$\texttt{fefa (s N)} \to \texttt{fefa N} \Leftarrow \emptyset \ \square\ \{\texttt{N} :: \texttt{nat}\}$$

Finally, let G_n be the simple goal $\emptyset \ \square\ \texttt{oh}\ (\texttt{s}^n\ \texttt{z}) \bowtie \texttt{ah}\ (\texttt{s}^n\ \texttt{z}) \ \square\ \emptyset \ \square\ \emptyset \ \square\ \emptyset$. Due to the behaviour of the function `halt` w.r.t. program $\mathcal{P}$, the following equivalences hold for any natural number n: $\varphi_n(n)\downarrow \iff \exists k \in \mathbb{N}$ such

that $\varphi_n(n)$ halts in k steps $\Longleftrightarrow$ solving G_n w.r.t. $\mathcal{P}$ using CLNC eventually leads to an opaque decomposition step.

In fact, in case that $\varphi_n(n)\downarrow$, solving G_n eventually leads to solving the goal `snd (tail [true])` $\bowtie$ `snd (tail [z])`, which needs opaque decomposition; and in the case that $\varphi_n(n)\downarrow$ does not halt, solving G_n gives rise to a non-terminating CLNC computation without opaque decompositions. With this we have shown that K can be reduced to the simple restriction of ODP, which is thus undecidable. □

6.6 Proofs of Results from Subsection 4.3

Proof of Lemma 15: Assume $TR(T, S_d) = (-, E)$. Since $\hat{G}$ is the type closure of G, we know that: (a) $\hat{\sigma}_t = mgu(S_t,\ E)$, (b) $\hat{T} = T\hat{\sigma}_t$. By Definition 12, $(\emptyset, (\hat{\sigma}_t, \sigma_d))$ is a well-typed solution of G and $\hat{G}$ iff the following four conditions are satisfied: (c) $\hat{\sigma}_t \in TSol(S_t)$, (d) $T\hat{\sigma}_t \vdash_{WT} X\sigma_d :: \tau\hat{\sigma}_t$, for all $(X :: \tau) \in T$, (c′) $\hat{\sigma}_t \in TSol(\hat{S}_t)$, (d′) $T\hat{\sigma}_t\hat{\sigma}_t \vdash_{WT} X\sigma_d :: \tau\hat{\sigma}_t\hat{\sigma}_t$, for all $(X :: \tau) \in T$. Note that (d′) can be stated like this because of (b). Moreover, (c) holds because of (a), (c′) holds because $\hat{S}_t$ is the set of equations in solved form representing $\hat{\sigma}_t$, and (d), (d′) are equivalent because $\hat{\sigma}_t$ is idempotent. In order to prove (d) we distinguish two cases:

$X \notin dom(\sigma_d)$: Then $X\sigma_d = X$ and (d) holds because of the type inference rule **VR**.

$X \in dom(\sigma_d)$: Then $X\sigma_d = t$ for some t such that $(X \approx t) \in S_d$. Because of Lemma 13, $T\hat{\sigma}_t \vdash_{WT} t :: (T\hat{\sigma}_t)(X)$. Since $t = X\sigma_d$ and $(T\hat{\sigma}_t)(X) = T(X)\hat{\sigma}_t = \tau\hat{\sigma}_t$, (d) is proved. □

Proof of Lemma 16: To prove the lemma, we consider the CLNC transformation rules one by one. In each case, we assume that G and G' are exactly as they appear in the presentation of the CLNC given in Subsection 4.2.

G' is admissible: For each CLNC rule, we can prove the preservation of the admissibility conditions **LN**, **EX**, **NC** and **SL** as in the first-order case [18].

In the sequel, we will use the notation $T_1 \uplus T_2 \uplus \ldots \uplus T_k$ to express the union of $k \geq 2$ type environments with pairwise disjoint domains.

Rules for the Unsolved Part

(ID) (a) Because of $(R, \theta) \in WTSol(G')$ we know:

1. $\theta_d \in Sol(S_d\rho_d, X \approx h\,\overline{V}_m)$ with $\rho_d = \{X \mapsto h\,\overline{V}_m\}$.
2. $dom(R) \subseteq (ran(\theta_d) \setminus dom((\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t)$.
3. $\mathcal{P} \vdash_{GORC} (P \;\square\; \overline{a}_p \asymp \overline{b}_p, C)\rho_d\theta_d$.
4. $\theta_t \in TSol(S'_t)$ with $\sigma'_t = mgu(\hat{S}_t,\ \tau \approx \tau')$, where $(h :: \overline{\tau}_m \to \tau) \in_{var} \Sigma$ and $\hat{T}(X) = \tau'$.
5. $((\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t\theta_t \cup R) \vdash_{WT} Y\theta_d :: ((\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t)(Y)\theta_t$ must hold, for all $Y \in dom((\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t)$.

Therefore, it holds that:

(D) $\rho_d\theta_d = \theta_d$ by 1.

(T) $\sigma'_t\theta_t = \theta_t$ and $\hat{\sigma}_t\sigma'_t = \sigma'_t$ by 4.

(E) Since the variables of $\{\overline{V}_m :: \overline{\tau}_m\}$ are fresh, $dom((\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t) = \{\overline{V}_m\} \uplus dom(T)$. Moreover, **(T)** implies that $\hat{T}\sigma'_t = T\sigma'_t$, $\hat{T}\sigma'_t\theta_t = T\theta_t$ and $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t\theta_t = \{\overline{V}_m :: \overline{\tau}_m\theta_t\} \uplus T\theta_t$.

Now we prove that $(\{\overline{V}_m :: \overline{\tau}_m\theta_t\} \cup R, \theta) \in WTSol(G)$:

- $\theta_d \in Sol(S_d)$, by 1 and **(D)**.
- $dom(\{\overline{V}_m :: \overline{\tau}_m\theta_t\} \cup R) \subseteq (ran(\theta_d) \setminus dom(T))$, by **(E)** and 2.
- $\mathcal{P} \vdash_{GORC} (P \;\square\; X\,\overline{a}_p \asymp X\,\overline{b}_p, C)\theta_d$: By 3 and **(D)**, for $i \in \{1, \ldots, p\}$ $\mathcal{P} \vdash_{GORC} (a_i \asymp b_i)\theta_d$, then $\mathcal{P} \vdash_{GORC} (h\,\overline{V}_m\,\overline{a}_p \asymp h\,\overline{V}_m\,\overline{b}_p)\theta_d$. Since $X\theta_d = X\rho_d\theta_d = (h\,\overline{V}_m)\theta_d$, we obtain that $\mathcal{P} \vdash_{GORC} (X\,\overline{a}_p \asymp X\,\overline{b}_p)\theta_d$.
- $\theta_t \in TSol(S_t)$, because 4 implies $\theta_t \in TSol(S'_t) \subseteq TSol(\hat{S}_t) \subseteq TSol(S_t)$.
- $(T\theta_t \cup \{\overline{V}_m :: \overline{\tau}_m\theta_t\} \cup R) \vdash_{WT} Y\theta_d :: T(Y)\theta_t$ for all $Y \in dom(T) \subseteq dom((\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t)$, by **(E)** and 5.

(b) To obtain that G' is also well-typed, we only need to prove:

[1] $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t \vdash_{WT} (X \approx h\,\overline{V}_m) :: \texttt{bool}$: Because of $(\overline{V}_m :: \overline{\tau}_m, \hat{T}) \vdash_{WT} X :: \tau'$, $(\overline{V}_m :: \overline{\tau}_m, \hat{T}) \vdash_{WT} h\,\overline{V}_m :: \tau$ and Lemma 4, $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t \vdash_{WT} X :: \tau'\sigma'_t$ and $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t \vdash_{WT} h\,\overline{V}_m :: \tau\sigma'_t$. Moreover, $\tau\sigma'_t = \tau'\sigma'_t$ by 4.

[2] $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t \vdash_{WT} (P, C, S_d)\rho_d :: \texttt{bool}$: This follows from $\hat{T} \vdash_{WT} (P, C, S_d) :: \texttt{bool}$ by applying Lemma 3 with $T_0 = \hat{T}$, $T_1 = (\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t$, $\sigma_t = \sigma'_t$, $\sigma_d = \rho_d$. This is possible by [1].

[3] For $i \in \{1, \ldots, p\}$ $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t \vdash_{WT} (a_i \asymp b_i)\rho_d :: \texttt{bool}$: Since G is well-typed, $\hat{T} \vdash_{WT} (X\,\overline{a}_p \asymp X\,\overline{b}_p) :: \texttt{bool}$. Thanks to [1], we can apply Lemma 3 with $T_0 = \hat{T}$, $T_1 = (\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t$, $\sigma_t = \sigma'_t$, $\sigma_d = \rho_d$ and we obtain $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t \vdash_{WT} (X\,\overline{a}_p \asymp X\,\overline{b}_p)\rho_d :: \texttt{bool}$, which can be rewritten as $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t \vdash_{WT} (h\,\overline{V}_m\,\overline{a}_p \asymp h\,\overline{V}_m\,\overline{b}_p)\rho_d :: \texttt{bool}$. Since we are assuming a transparent step, h must be $(m+p)$-transparent. Then, we can apply Lemma 7 to obtain types μ_i, $i \in \{1, \ldots, p\}$ such that $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t \vdash_{WT} a_i\rho_d :: \mu_i :: b_i\rho_d$.

(DC1) (a) Because of $(R, \theta) \in \mathit{WTSol}(G')$ we know:

1. $\theta_d \in \mathit{Sol}(S_d)$.
2. $\mathit{dom}(R) \subseteq (\mathit{ran}(\theta_d) \setminus \mathit{dom}(T))$.
3. $\mathcal{P} \vdash_{\mathit{GORC}} (P \;\square\; \overline{a}_m \asymp \overline{b}_m, C)\theta_d$.
4. $\theta_t \in \mathit{TSol}(S_t)$.
5. $(T\theta_t \cup R) \vdash_{WT} Y\theta_d :: T(Y)\theta_t$ must hold, for all $Y \in \mathit{dom}(T)$.

$(R, \theta) \in \mathit{WTSol}(G)$: $\mathcal{P} \vdash_{\mathit{GORC}} (P \;\square\; h\,\overline{a}_m \asymp h\,\overline{b}_m, C)\theta_d$ follows easily from 3. Moreover, S_t, S_d and T are not modified.

(b) G well-typed implies, in particular, that $\hat{T} \vdash_{WT} (P, C, S_d) :: \texttt{bool}$ and $\hat{T} \vdash_{WT} (h\overline{a}_m \asymp h\overline{b}_m) :: \texttt{bool}$. Therefore, G' is well-typed, if we can prove that $\hat{T} \vdash_{WT} (a_i \asymp b_i) :: \texttt{bool}$, for $i \in \{1, \ldots, m\}$. Since $\hat{T} \vdash_{WT} (h\,\overline{a}_m \asymp h\,\overline{b}_m) :: \texttt{bool}$ and we are assuming a transparent step, h must be m-transparent. By Lemma 7, there exist types τ_i for all $i \in \{1, \ldots, m\}$ such that $\hat{T} \vdash_{WT} a_i :: \tau_i :: b_i$, which implies $\hat{T} \vdash_{WT} (a_i \asymp b_i) :: \texttt{bool}$.

(BD) (a) Because of $(R, \theta) \in \mathit{WTSol}(G')$ we know:

1. $\theta_d \in \mathit{Sol}(S_d\rho_d, X \approx s)$ with $\rho_d = \{X \mapsto s\}$.
2. $\mathit{dom}(R) \subseteq (\mathit{ran}(\theta_d) \setminus \mathit{dom}(\hat{T}\sigma'_t)$.
3. $\mathcal{P} \vdash_{\mathit{GORC}} (P \;\square\; \overline{a}_k \asymp \overline{b}_k, C)\rho_d\theta_d$.

4. $\theta_t \in TSol(S'_t)$, $\sigma'_t = mgu(\hat{S}_t,\ \tau \approx \tau')$ with $PA(\hat{T}, s\ \overline{b}_k) = (s^{\tau'} -)^-$, $\hat{T}(X) = \tau$.

5. $(\hat{T}\sigma'_t\theta_t \cup R) \vdash_{WT} Y\theta_d :: (\hat{T}\sigma'_t)(Y)\theta_t$ must hold, for all $Y \in dom(\hat{T}\sigma'_t)$.

Therefore, it holds that:

(D) $\rho_d\theta_d = \theta_d$ by 1.

(T) $\sigma'_t\theta_t = \theta_t$ and $\hat{\sigma}_t\sigma'_t = \sigma'_t$ by 4.

(E) $\hat{T}\sigma'_t = T\sigma'_t$ and $\hat{T}\sigma'_t\theta_t = T\theta_t$ using **(T)**.

Now we prove that $(R, \theta) \in WTSol(G)$:

- $\theta_d \in Sol(S_d)$, by 1 and **(D)**.
- $dom(R) \subseteq ran(\theta_d) \setminus dom(T)$, by **(E)** and 2.
- $\mathcal{P} \vdash_{GORC} (P \ \square\ X\ \overline{a}_k \asymp s\ \overline{b}_k, C)\theta_d$: By 3 and **(D)**, for $i \in \{1, \ldots, k\}$ $\mathcal{P} \vdash_{GORC} (a_i \asymp b_i)\theta_d$, then $\mathcal{P} \vdash_{GORC} (s\ \overline{a}_k \asymp s\ \overline{b}_k)\theta_d$. Since $X\theta_d = X\rho_d\theta_d = s\theta_d$, we obtain that $\mathcal{P} \vdash_{GORC} (X\ \overline{a}_k \asymp s\ \overline{b}_k)\theta_d$.
- $\theta_t \in TSol(S_t)$, because 4 implies $TSol(S'_t) \subseteq TSol(\hat{S}_t) \subseteq TSol(S_t)$.
- $(T\theta_t \cup R) \vdash_{WT} Y\theta_d :: T(Y)\theta_t$ for all $Y \in dom(T)$, because of **(E)** and 5.

(b) To obtain that G' is also well-typed, we only need to prove:

[1] $\hat{T}\sigma'_t \vdash_{WT} (X \approx s) :: \texttt{bool}$: Because of $\hat{T} \vdash_{WT} X :: \tau$, $\hat{T} \vdash_{WT} s :: \tau'$ and Lemma 4, $\hat{T}\sigma'_t \vdash_{WT} X :: \tau\sigma'_t$ and $\hat{T}\sigma'_t \vdash_{WT} s :: \tau'\sigma'_t$. Moreover, $\tau\sigma'_t = \tau'\sigma'_t$ by 4.

[2] $\hat{T}\sigma'_t \vdash_{WT} (P, C, S_d)\rho_d :: \texttt{bool}$: This follows from $\hat{T} \vdash_{WT} (P, C, S_d) :: \texttt{bool}$ by applying the Lemma 3 with $T_0 = \hat{T}$, $T_1 = \hat{T}\sigma'_t$, $\sigma_t = \sigma'_t$, $\sigma_d = \rho_d$. This is possible by [1].

[3] For $i \in \{1, \ldots, k\}$, $\hat{T}\sigma'_t \vdash_{WT} (a_i \asymp b_i)\rho_d :: \texttt{bool}$: Since G is well-typed, $\hat{T} \vdash_{WT} (X\ \overline{a}_k \asymp s\ \overline{b}_k) :: \texttt{bool}$. Applying Lemma 3 as in [2], we can deduce $\hat{T}\sigma'_t \vdash_{WT} (X\ \overline{a}_k \asymp s\ \overline{b}_k)\rho_d :: \texttt{bool}$, which can be rewritten as $\hat{T}\sigma'_t \vdash_{WT} (s\overline{a}_k \asymp s\overline{b}_k)\rho_d :: \texttt{bool}$, where s must have the form $h\overline{t}_m$ due to the assumptions of **(BD)**. Since we are assuming a transparent step, h must be $(m + k)$-transparent. Then, we can apply Lemma 7 to obtain types τ_i, $i \in \{1, \ldots, k\}$ such that $\hat{T}\sigma'_t \vdash_{WT} a_i\rho_d :: \tau_i :: b_i\rho_d$.

(IM) (a) Because of $(R, \theta) \in WTSol(G')$ we know:

1. $\theta_d \in Sol(S_d\rho_d, X \approx h\,\overline{V}_m)$ with $\rho_d = \{X \mapsto h\,\overline{V}_m\}$.
2. $dom(R) \subseteq ran(\theta_d) \setminus dom((\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t)$.
3. $\mathcal{P} \vdash_{GORC} (P \;\square\; \overline{V}_m\,\overline{a}_k \asymp \overline{e}_m\,\overline{b}_k, C)\rho_d\theta_d$.
4. $\theta_t \in TSol(S'_t)$, with $\sigma'_t = mgu(\hat{S}_t,\ \tau \approx \tau')$, where $(h :: \overline{\tau}_m \to \tau) \in_{var} \Sigma$ and $\hat{T}(X) = \tau'$.
5. $((\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t\theta_t \cup R) \vdash_{WT} Y\theta_d :: ((\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t)(Y)\theta_t$ must hold, for all $Y \in dom((\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t)$.

Therefore, it holds that:

(D) $\rho_d\theta_d = \theta_d$ by 1.

(T) $\sigma'_t\theta_t = \theta_t$ and $\hat{\sigma}_t\sigma'_t = \sigma'_t$ by 4.

(E) Since the variables of $\{\overline{V}_m :: \overline{\tau}_m\}$ are fresh, $dom((\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t) = \{\overline{V}_m\} \uplus dom(T)$. Moreover, **(T)** implies that $\hat{T}\sigma'_t = T\sigma'_t$, $\hat{T}\sigma'_t\theta_t = T\theta_t$ and $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t\theta_t = \{\overline{V}_m :: \overline{\tau}_m\theta_t\} \uplus T\theta_t$.

Now we prove that $(\{\overline{V}_m :: \overline{\tau}_m\theta_t\} \cup R, \theta) \in WTSol(G)$:

- $\theta_d \in Sol(S_d)$, by 1 and **(D)**.
- $dom(\{\overline{V}_m :: \overline{\tau}_m\theta_t\} \cup R) \subseteq ran(\theta_d) \setminus dom(T)$, by **(E)** and 2.
- $\mathcal{P} \vdash_{GORC} (P \;\square\; X\,\overline{a}_k \asymp h\,\overline{e}_m\,\overline{b}_k, C)\theta_d$: By 3 and **(D)**, for $i \in \{1, \ldots, k\}$ $\mathcal{P} \vdash_{GORC} (a_i \asymp b_i)\theta_d$; for $j \in \{1, \ldots, m\}$ $\mathcal{P} \vdash_{GORC} (V_j \asymp e_j)\theta_d$, then $\mathcal{P} \vdash_{GORC} (h\,\overline{V}_m\,\overline{a}_k \asymp h\,\overline{e}_m\,\overline{b}_k)\theta_d$. Since $X\theta_d = X\rho_d\theta_d = (h\,\overline{V}_m)\theta_d$, we obtain that $\mathcal{P} \vdash_{GORC} (X\,\overline{a}_k \asymp h\,\overline{e}_m\,\overline{b}_k)\theta_d$.
- $\theta_t \in TSol(S_t)$, because 4 implies $TSol(S'_t) \subseteq TSol(\hat{S}_t) \subseteq TSol(S_t)$.
- $(T\theta_t \cup \{\overline{V}_m :: \overline{\tau}_m\theta_t\} \cup R) \vdash_{WT} Y\theta_d :: T(Y)\theta_t$ for all $Y \in dom(T) \subseteq dom((\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t)$, by **(E)** and 5.

(b) To obtain that G' is also well-typed, we only need to prove:

[1] $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t \vdash_{WT} (X \approx h\,\overline{V}_m) :: \texttt{bool}$: Because of $(\overline{V}_m :: \overline{\tau}_m, \hat{T}) \vdash_{WT} X :: \tau'$, $(\overline{V}_m :: \overline{\tau}_m, \hat{T}) \vdash_{WT} h\,\overline{V}_m :: \tau$ and Lemma 4, $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t \vdash_{WT} X :: \tau'\sigma'_t$ and $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t \vdash_{WT} h\,\overline{V}_m :: \tau\sigma'_t$. Moreover, $\tau\sigma'_t = \tau'\sigma'_t$ by 4.

[2] $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t \vdash_{WT} (P, C, S_d)\rho_d :: \texttt{bool}$: Analogous to [2] of rule **(BD)**.

[3] For $i \in \{1, \ldots, k\}$ $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t \vdash_{WT} (a_i \asymp b_i)\rho_d :: \texttt{bool}$ and for $j \in \{1, \ldots, m\}$ $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t \vdash_{WT} (V_j \asymp e_j)\rho_d :: \texttt{bool}$: Since G is well-typed, $\hat{T} \vdash_{WT} (X\,\overline{a}_k \asymp h\overline{e}_m\overline{b}_k) :: \texttt{bool}$. Thanks to [1], we can apply Lemma 3 with $T_0 = \hat{T}$, $T_1 = (\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t$, $\sigma_t = \sigma'_t$, $\sigma_d = \rho_d$ and we obtain $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t \vdash_{WT} (X\,\overline{a}_k \asymp h\,\overline{e}_m\,\overline{b}_k)\rho_d :: \texttt{bool}$, which can be rewritten as $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t \vdash_{WT} (h\,\overline{V}_m\,\overline{a}_k \asymp h\,\overline{e}_m\,\overline{b}_k)\rho_d :: \texttt{bool}$. Since we are assuming a transparent step, h must be $(m+k)$-transparent. Then, we can apply Lemma 7 to obtain types μ_i, $i \in \{1, \ldots, k\}$ and ν_j, $j \in \{1, \ldots, m\}$ such that $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t \vdash_{WT} a_i\rho_d :: \mu_i :: b_i\rho_d$ and $(\overline{V}_m :: \overline{\tau}_m, \hat{T})\sigma'_t \vdash_{WT} V_j\rho_d :: \nu_j :: e_j\rho_d$. Moreover, $V_j\rho_d = V_j$, for all $j \in \{1, \ldots, m\}$.

(NR1) (a) Because of $(R, \theta) \in WTSol(G')$ we know:

1. $\theta_d \in Sol(S_d)$.
2. $dom(R) \subseteq ran(\theta_d) \setminus dom(T \cup T')$ and $(f\,\overline{t}_n \to r \Leftarrow C' \;\square\; T') \in_{var} \mathcal{P}$.
3. $\mathcal{P} \vdash_{GORC} (\overline{e}_n \to \overline{t}_n, P \;\square\; C', r\overline{a}_k \asymp b, C)\theta_d$.
4. $\theta_t \in TSol(S_t)$.
5. $((T \cup T')\theta_t \cup R) \vdash_{WT} Y\theta_d :: (T \cup T')(Y)\theta_t$ must hold, for all $Y \in dom(T \cup T')$.

Since all the variables in T' are fresh, it holds that:

(E) $T \cup T' = T \uplus T'$ and $(T \cup T')\theta_t = (T\theta_t \uplus T'\theta_t)$.

Now we prove that $(T'\theta_t \cup R, \theta) \in WTSol(G)$. Since S_t and S_d are not modified, we only need to prove that:

- $dom(T'\theta_t \cup R) \subseteq (ran(\theta_d) \setminus dom(T))$, by **(E)** and 2.

- $(T\theta_t \cup T'\theta_t \cup R) = ((T \cup T')\theta_t \cup R) \vdash_{WT} Y\theta_d :: T(Y)\theta_t$ for all $Y \in dom(T) \subseteq dom(T \cup T')$, by **(E)** and 5.

- $\mathcal{P} \vdash_{GORC} (P \;\square\; f\,\overline{e}_n\,\overline{a}_k \asymp b, C)\theta_d$. $\mathcal{P} \vdash_{GORC} P\theta_d$ and $\mathcal{P} \vdash_{GORC} C\theta_d$ are obvious by 3. Moreover, 3 also implies that (a): $\mathcal{P} \vdash_{GORC} (e_i \to t_i)\theta_d$, (b): $\mathcal{P} \vdash_{GORC} C'\theta_d$ and (c): $\mathcal{P} \vdash_{GORC} (r\,\overline{a}_k \asymp b)\theta_d \iff$ There exists a total pattern t such that (d): $\mathcal{P} \vdash_{GORC} (r\,\overline{a}_k)\theta_d \to t$ and (e): $\mathcal{P} \vdash_{GORC} b\theta_d \to t$. With (a), (b) and (d) we obtain that $\mathcal{P} \vdash_{GORC} (f\overline{e}_n\overline{a}_k)\theta_d \to t$, this fact and (e) implies that $\mathcal{P} \vdash_{GORC} (f\,\overline{e}_n\,\overline{a}_k \asymp b)\theta_d$.

(b) $(f\,\overline{t}_n \to r \Leftarrow C' \;\square\; T') \in_{var} \mathcal{P}$ implies that $T' \vdash_{WT} t_i :: \tau_i$, for $i \in \{1,\ldots,n\}$, $T' \vdash_{WT} r :: \tau$, $T' \vdash_{WT} C' ::$ `bool` and $T' \vdash_{WT} f :: \overline{\tau}_n \to \tau$. To prove that G' is also well-typed, we apply Lemma 13(b), finding $\lambda_t \in TSub$ such that $(T'\lambda_t, \hat{T}) \vdash_{WT} G'$.

[1] Independently of the choice of λ_t, $(T'\lambda_t, \hat{T}) \vdash_{WT} (P, C, C', S_d) ::$ `bool` follows from Lemma 4, because G and $(f\,\overline{t}_n \to r \Leftarrow C' \;\square\; T')$ are well-typed.

[2] We still have to deal with the rest of G', namely $e_i \to t_i$, for $i \in \{1,\ldots,n\}$ and $r\,\overline{a}_k \asymp b$. Because $\hat{T} \vdash_{WT} G$, we know that $\hat{T} \vdash_{WT} (f\,\overline{e}_n\,\overline{a}_k \asymp b) ::$ `bool`, and there must exist types ν_i, μ_j, μ such that $\hat{T} \vdash_{WT} e_i :: \nu_i$, for $i \in \{1,\ldots,n\}$, $\hat{T} \vdash_{WT} a_j :: \mu_j$, for $j \in \{1,\ldots,k\}$, $\hat{T} \vdash_{WT} b :: \mu$ and $(\overline{\nu}_n \to \overline{\mu}_k \to \mu) \geq \overline{\tau}_n \to \tau$. We choose $\lambda_t \in TSub$ such that $(\overline{\tau}_n \to \tau)\lambda_t = (\overline{\nu}_n \to \overline{\mu}_k \to \mu)$. Therefore, $\nu_i = \tau_i\lambda_t$, for $i \in \{1,\ldots,n\}$, and $\tau\lambda_t = (\overline{\mu}_k \to \mu)$. Due to Lemma 4, $(T'\lambda_t, \hat{T}) \vdash_{WT} t_i :: \tau_i\lambda_t = \nu_i$, and hence $(T'\lambda_t, \hat{T}) \vdash_{WT} (e_i \to t_i) ::$ `bool`, for $i \in \{1,\ldots,n\}$. Also because of Lemma 4, $(T'\lambda_t, \hat{T}) \vdash_{WT} r :: \tau\lambda_t = (\overline{\mu}_k \to \mu)$, which implies $(T'\lambda_t, \hat{T}) \vdash_{WT} r\,\overline{a}_k :: \mu :: b$ and hence $(T'\lambda_t, \hat{T}) \vdash_{WT} (r\,\overline{a}_k \asymp b) ::$ `bool`.

(GN) (a) Because of $(R, \theta) \in WTSol(G')$ we know:

1. $\theta_d \in Sol(S_d\rho_d, X \approx f\,\overline{t}_p)$, with $\rho_d = \{X \mapsto f\,\overline{t}_p\}$.

2. $dom(R) \subseteq ran(\theta_d) \setminus dom((\hat{T} \cup T')\sigma'_t)$ and $(f\,\overline{t}_p\,\overline{s}_q \to r \Leftarrow C' \;\square\; T') \in_{var} \mathcal{P}$.

3. $\mathcal{P} \vdash_{GORC} (\overline{e}_q \to \overline{s}_q, P \;\square\; C', r\,\overline{a}_k \asymp b, C)\rho_d\theta_d$.

4. $\theta_t \in TSol(S'_t)$ with $\sigma'_t = mgu(\hat{S}_t,\ \tau' \approx \overline{\tau}_q \to \tau)$ where $\tau' = \hat{T}(X)$ and $(f :: \overline{\tau}_p \to \overline{\lambda}_q \to \tau) \in_{var} \Sigma$.

5. $((\hat{T} \cup T')\sigma'_t\theta_t \cup R) \vdash_{WT} Y\theta_d :: (\hat{T} \cup T')(Y)\sigma'_t\theta_t$ must hold, for all $Y \in dom((\hat{T} \cup T')\sigma'_t)$.

Therefore, it holds that:

(D) $\rho_d\theta_d = \theta_d$ by 1.

(T) $\sigma'_t\theta_t = \theta_t$ and $\hat{\sigma}_t\sigma'_t = \sigma'_t$ by 4.

(E) Since all the variables in T' are fresh, $\hat{T} \cup T' = T \uplus T'$. Moreover, **(T)** implies that $(\hat{T} \cup T')\sigma'_t = (T \cup T')\sigma'_t$ and $(\hat{T} \cup T')\sigma'_t\theta_t = (T\theta_t \cup T'\theta_t)$.

Now we prove that $(T'\theta_t \cup R, \theta) \in WTSol(G)$:

- $\theta_d \in Sol(S_d)$, by 1 and **(D)**.
- $dom(T'\theta_t \cup R) \subseteq (ran(\theta_d) \setminus dom(T))$, by **(E)** and 2.
- $\mathcal{P} \vdash_{GORC} (P \ \square\ X\,\overline{e}_q\,\overline{a}_k \asymp b, C)\theta_d$: By 3 and **(D)**, we have (a): $\mathcal{P} \vdash_{GORC} C'\theta_d$, (b): $\mathcal{P} \vdash_{GORC} (e_j \to s_j)\theta_d$, for $j \in \{1, \ldots, q\}$, and (c): $\mathcal{P} \vdash_{GORC} (r\,\overline{a}_k \asymp b)\theta_d \iff$ There exists a total pattern t such that (d): $\mathcal{P} \vdash_{GORC} (r\ \overline{a}_k)\theta_d \to t$ and (e): $\mathcal{P} \vdash_{GORC} b\theta_d \to t$. By (a), (b), (d) and $X\theta_d = X\rho_d\theta_d = (f\ \overline{t}_p)\theta_d$, we have that $\mathcal{P} \vdash_{GORC} (X\ \overline{e}_q\ \overline{a}_k)\theta_d \to t$. Therefore, by (e) we conclude that $\mathcal{P} \vdash_{GORC} (X\ \overline{e}_q\ \overline{a}_k \asymp b)\theta_d$.
- $\theta_t \in TSol(S_t)$, because 4 implies $TSol(S'_t) \subseteq TSol(\hat{S}_t) \subseteq TSol(S_t)$.
- $(T\theta_t \cup T'\theta_t \cup R) \vdash_{WT} Y\theta_d :: T(Y)\theta_t$ for all $Y \in dom(T) \subseteq dom(\hat{T} \cup T')$, because **(E)** and 5.

(b) Since $(f\ \overline{t}_p\ \overline{s}_q \to r \Leftarrow C' \ \square\ T') \in_{var} \mathcal{P}$ is a fresh variant of a well-typed f-rule and $(f :: \overline{\tau}_p \to \overline{\lambda}_q \to \tau) \in_{var} \Sigma$, we have that for $l \in \{1, \ldots, p\}$: $T' \vdash_{WT} t_l :: \tau_l$; for $j \in \{1, \ldots, q\}$: $T' \vdash_{WT} s_j :: \lambda_j$; $T' \vdash_{WT} r :: \tau$ and $T' \vdash_{WT} C' :: \texttt{bool}$. To obtain that G' is also well-typed, we need to prove:

[1] $(T', \hat{T})\sigma'_t \vdash_{WT} (X \approx f\ \overline{t}_p) :: \texttt{bool}$: Because of $\hat{T} \vdash_{WT} X :: \tau'$ and $T' \vdash_{WT} f\ \overline{t}_p :: (\overline{\lambda}_q \to \tau)$ and Lemma 4, $(T', \hat{T})\sigma'_t \vdash_{WT} X :: \tau'\sigma'_t$ and $(T', \hat{T})\sigma'_t \vdash_{WT} f\ \overline{t}_p :: (\overline{\lambda}_q \to \tau)\sigma'_t$. Moreover, $\tau'\sigma'_t = (\overline{\lambda}_q \to \tau)\sigma'_t$ by 4.

[2] $(T', \hat{T})\sigma'_t \vdash_{WT} (P, C', C, S_d)\rho_d :: \texttt{bool}$: This follows by Lemma 3 from $(T', \hat{T}) \vdash_{WT} (P, C, C', S_d) :: \texttt{bool}$, which can be applied because of [1].

[3] Because $\hat{T} \vdash_{WT} G$, we know that $\hat{T} \vdash_{WT} (X\ \overline{e}_q\ \overline{a}_k \asymp b) :: \texttt{bool}$, and there must exist types ν_i, μ_j, μ such that $\hat{T} \vdash_{WT} e_i :: \nu_i$, for $i \in \{1, \ldots, q\}$, $\hat{T} \vdash_{WT} a_j :: \mu_j$, for $j \in \{1, \ldots, k\}$, $\hat{T} \vdash_{WT} b :: \mu$ and $(\overline{\nu}_q \to \overline{\mu}_k \to \mu) = \hat{T}(X) = \tau'$. By 4, $\tau'\sigma'_t = (\overline{\lambda}_q \to \tau)\sigma'_t$, i.e., $(\overline{\nu}_q \to \overline{\mu}_k \to \mu)\sigma'_t = (\overline{\lambda}_q \to \tau)\sigma'_t$. Therefore, $\nu_i\sigma'_t = \lambda_i\sigma'_t$, for $i \in \{1, \ldots, q\}$, and $\tau\sigma'_t = (\overline{\mu}_k \to \mu)\sigma'_t$. Using Lemma 3 (which can be applied thanks to [1]) we get $(T', \hat{T})\sigma'_t \vdash_{WT} e_i\rho_d :: \nu_i\sigma'_t$, $(T', \hat{T})\sigma'_t \vdash_{WT} s_i\rho_d :: \lambda_i\sigma'_t$ for $i \in \{1, \ldots, q\}$. Since $\nu_i\sigma'_t = \lambda_i\sigma'_t$, we obtain $(T', \hat{T})\sigma'_t \vdash_{WT} (e_i \to s_i)\rho_d :: \texttt{bool}$. Applying again Lemma 3, we get now $(T', \hat{T})\sigma'_t \vdash_{WT} a_j\rho_d :: \mu_j\sigma'_t$, $(T', \hat{T})\sigma'_t \vdash_{WT} b\rho_d :: \mu\sigma'_t$ and $(T', \hat{T})\sigma'_t \vdash_{WT} r\rho_d :: \tau\sigma'_t = (\overline{\mu}_k \to \mu)\sigma'_t$. Hence $(T', \hat{T})\sigma'_t \vdash_{WT} (r\ \overline{a}_k \asymp b)\rho_d :: \texttt{bool}$.

(GD) (a) Because of $(R, \theta) \in WTSol(G')$ we know:

1. $\theta_d \in Sol(S_d\rho_d, X \approx h\ \overline{V}_{m-p}, Y \approx h\ \overline{W}_{m-q})$ where $\rho_d = \{X \mapsto h\ \overline{V}_{m-p}, Y \mapsto h\ \overline{W}_{m-q}\}$.
2. $dom(R) \subseteq ran(\theta_d) \setminus dom((\{\overline{V}_{m-p} :: \overline{\tau'}_{m-p}\} \cup \{\overline{W}_{m-q} :: \overline{\tau''}_{m-q}\} \cup \hat{T})\sigma'_t)$.
3. $\mathcal{P} \vdash_{GORC} (P \ \square\ \overline{V}_{m-p}\ \overline{a}_p \asymp \overline{W}_{m-q}\ \overline{b}_q, C)\rho_d\theta_d$.
4. $\theta_t \in TSol(S'_t)$ with $\sigma'_t = mgu(\hat{S}_t,\ \tau' \approx \overline{\lambda'}_p \to \tau'_0,\ \tau'' \approx \overline{\lambda''}_q \to \tau''_0)$, where $\tau' = T(X), \tau'' = T(Y)$, $(h :: \overline{\tau'}_{m-p} \to \overline{\lambda'}_p \to \tau'_0) \in_{var} \Sigma$ and $(h :: \overline{\tau''}_{m-q} \to \overline{\lambda''}_q \to \tau''_0) \in_{var} \Sigma$.
5. $(T_1\sigma'_t\theta_t \cup R) \vdash_{WT} Z\theta_d :: T_1(Z)\sigma'_t\theta_t$ must hold, for all $Z \in dom(T_1\sigma'_t)$, where $T_1 := \{\overline{V}_{m-p} :: \overline{\tau'}_{m-p}\} \cup \{\overline{W}_{m-q} :: \overline{\tau''}_{m-q}\} \cup \hat{T}$.

Therefore, it holds that:

(D) $\rho_d\theta_d = \theta_d$ by 1.

(T) $\sigma'_t\theta_t = \theta_t$ and $\hat{\sigma}_t\sigma'_t = \sigma'_t$ by 4.

(E) Since the variables of $\{\overline{V}_{m-p} :: \overline{\tau'}_{m-p}\} \cup \{\overline{W}_{m-q} :: \overline{\tau''}_{m-q}\}$ are fresh, $dom(T_1) = \{\overline{V}_{m-p}\} \uplus \{\overline{W}_{m-q}\} \uplus dom(T)$. Moreover, by **(T)**, $T_1\theta_t = \{\overline{V}_{m-p} :: \overline{\tau'}_{m-p}\theta_t\} \uplus \{\overline{W}_{m-q} :: \overline{\tau''}_{m-q}\theta_t\} \uplus T\theta_t$ and $\hat{T}\sigma'_t = T\sigma'_t$.

We prove that $(\{\overline{V}_{m-p} :: \overline{\tau'}_{m-p}\theta_t\} \cup \{\overline{W}_{m-q} :: \overline{\tau''}_{m-q}\theta_t\} \cup R, \theta) \in \ WTSol(G)$:

- $\theta_d \in Sol(S_d)$, by 1 and **(D)**.
- $dom(\{\overline{V}_{m-p} :: \overline{\tau'}_{m-p}\theta_t\} \cup \{\overline{W}_{m-q} :: \overline{\tau''}_{m-q}\theta_t\} \cup R) \subseteq (ran(\theta_d) \setminus dom(T))$, by **(E)** and 2.
- $\mathcal{P} \vdash_{GORC} (P \ \square \ X\,\overline{a}_p \asymp Y\,\overline{b}_q, C)\theta_d$: $\mathcal{P} \vdash_{GORC} (h\,\overline{V}_{m-p}\,\overline{a}_p \asymp h\,\overline{W}_{m-q}\,\overline{b}_q)\theta_d$ because of 3 and **(D)**, but $X\theta_d = X\rho_d\theta_d = (h\,\overline{V}_{m-p})\theta_d$ and $Y\theta_d = Y\rho_d\theta_d = (h\,\overline{W}_{m-q})\theta_d$, therefore $\mathcal{P} \vdash_{GORC} (X\,\overline{a}_p \asymp Y\,\overline{b}_q)\theta_d$.
- $\theta_t \in TSol(S_t)$, because 4 implies $TSol(S'_t) \subseteq TSol(\hat{S}_t) \subseteq TSol(S_t)$.
- Since $(T\theta_t \cup \{\overline{V}_{m-p} :: \overline{\tau'}_{m-p}\theta_t\} \cup \{\overline{W}_{m-q} :: \overline{\tau''}_{m-q}\theta_t\} \cup R) = (T_1\theta_t \cup R)$, then $(T_1\theta_t \cup R) \vdash_{WT} Z\theta_d :: T(Z)\theta_t$ for all $Z \in dom(T) \subseteq dom(T1)$, because of **(E)** and 5.

(b) Let us assume $p \geq q$; the case $p \leq q$ is symmetrical. To show that G' is well-typed, we prove [1], [2], [3] and [4].

[1] $T_1\sigma'_t \vdash_{WT} (X \approx h\,\overline{V}_{m-p}) :: \texttt{bool}$: Using $T_1 \vdash_{WT} h\,\overline{V}_{m-p} :: (\overline{\lambda'}_p \to \tau'_0)$, $T_1 \vdash_{WT} X :: \tau'$ and Lemma 4, we obtain $T_1\sigma'_t \vdash_{WT} X :: \tau'\sigma'_t$ and $T_1\sigma'_t \vdash_{WT} h\,\overline{V}_{m-p} :: (\overline{\lambda'}_p \to \tau'_0)\sigma'_t$. Moreover, $\tau'\sigma'_t = (\overline{\lambda'}_p \to \tau'_0)\sigma'_t$ by 4.

[2] $T_1\sigma'_t \vdash_{WT} (Y \approx h\,\overline{W}_{m-q}) :: \texttt{bool}$: The proof is similar to [1], using the facts $T_1\sigma'_t \vdash_{WT} Y :: \tau''\sigma'_t$ and $T_1\sigma'_t \vdash_{WT} h\,\overline{W}_{m-q} :: (\overline{\lambda''}_q \to \tau''_0)\sigma'_t$ and $\tau''\sigma'_t = (\overline{\lambda''}_q \to \tau''_0)\sigma'_t$.

[3] $T_1\sigma'_t \vdash_{WT} (P, C, S_d)\rho_d :: \texttt{bool}$: This follows from $T_1 \vdash_{WT} (P, C, S_d) :: \texttt{bool}$, by Lemma 3, which can be applied because of [1], [2].

[4] $T_1\sigma'_t \vdash_{WT} (\overline{V}_{m-p}\,\overline{a}_p \asymp \overline{W}_{m-q}\,\overline{b}_q)\rho_d :: \texttt{bool}$: We show this by proving three things:

(a) $T_1\sigma'_t \vdash_{WT} (V_j \asymp W_j)\rho_d :: \texttt{bool}$, $j \in \{1, \ldots, (m-p)\}$.

(b) $T_1\sigma'_t \vdash_{WT} (a_i \asymp W_{m-p+i})\rho_d :: \texttt{bool}$, $i \in \{1, \ldots, (p-q)\}$.

(c) $T_1\sigma'_t \vdash_{WT} (a_{p-q+k} \asymp b_k)\rho_d :: \texttt{bool}$, $k \in \{1, \ldots, q\}$.

Proof of (a), (b), (c): Since G is well-typed, $T_1 \vdash_{WT} (X\,\overline{a}_p \asymp Y\,\overline{b}_q) :: \texttt{bool}$. Using Lemma 3 (which is applicable because of [1], [2]) we obtain $T_1\sigma'_t \vdash_{WT} (X\,\overline{a}_p \asymp Y\,\overline{b}_q)\rho_d :: \texttt{bool}$, which can be rewritten as $T_1\sigma'_t \vdash_{WT}$

$(h\overline{V}_{m-p}\overline{a}_p \asymp h\overline{W}_{m-q}\overline{b}_q)\rho_d :: \texttt{bool}$. Since we are assuming a transparent CLNC step, h must be m-transparent. Then, we can apply Lemma 7 to obtain types ν_j, for $j \in \{1,\ldots,(m-p)\}$, μ_i, for $i \in \{1,\ldots,(p-q)\}$ and λ_k, for $k \in \{1,\ldots,q\}$ such that $T_1\sigma'_t \vdash_{WT} V_j\ \rho_d :: \nu_j :: W_j\rho_d$, $T_1\sigma'_t \vdash_{WT} a_i\rho_d :: \mu_i :: W_{m-p+i}\rho_d$ and $T_1\sigma'_t \vdash_{WT} a_{p-q+k}\rho_d :: \lambda_k :: b_k\rho_d$.

(CF1), (CY) Analogously to [18], it is easy to prove that $Sol(G) = \emptyset$ whenever these rules are applicable.

Rules for the Delayed Part

We prove only the correctness of **(EL)**. For the other rules in this part, the proofs of related rules in the solved part can be easily adapted.

(EL) (a) It is clear that $(R,\theta') \in WTSol(G)$, where $\theta'_t = \theta_t$, $X\theta'_d = \bot$ and $Y\theta'_d = Y\theta_d$ for all variables $Y \neq X$. Note that $X\theta'_d = \bot$ is allowed because X is a produced variable in G. Being produced, X is also existential, and then $(R,\theta') \in WTSol(G)$ implies $(R,\theta) \in_{ex} WTSol(G)$.

(b) If G is well-typed then G' is also well-typed. □

Proof of Lemma 17: We organize our reasoning in two stages. First, we use the fact that $\mathcal{M}$ is a witness of $(R,\theta) \in TASol(\hat{G})$ to find a CLNC rule TR which acts as a candidate to transform G into G' as required by the lemma. This part of the proof ignores types and is performed by a case analysis. For each candidate rule TR, the second stage of our reasoning uses the type information present in $\mathcal{M}$ to show that TR can be actually applied to G (which may involve dynamic type checking) and to build a smaller type-annotated $\mathcal{M}' \lhd \mathcal{M}$ witnessing $(R',\theta') \in TASol(\hat{G}')$, for some suitable $(R',\theta') \approx_{ex} (R,\theta)$. In each case, θ' and $\mathcal{M}'$ must be built explicitly, but the construction of R' and the reasoning which proves G' well-typed can be left implicit. This is because the type annotation within $\mathcal{M}'$ implies well-typedness of G', and R' can be defined as the set of all type assumptions $X :: \tau$ such that X^τ occurs in $\mathcal{M}'$, but $X \notin dom(\hat{\mathrm{T}}'\theta'_t \cup R)$.

In what follows, items **1**, **2**, etc. enumerate the cases in our main case distinction. Subcases are indicated by a similar notation, as **1.2**, **1.2.1**, etc. When performing the second stages of our reasoning for each particular CLNC rule TR, we will assume that G and G' are written with exactly the same notation used in the presentation of CLNC in Subsection 4.2. In several

cases, we will leave out proof details which are easy to complete, or similar to the reasonings already performed for some other cases.

Let us now start our case analysis. By the One-step Soundness Lemma 16, we can ignore the failure rules, because G is solvable. Moreover, we can assume that the $(P \;\square\; C)$ part of G is nor empty, because G is not solved.

1 Assume that G satisfies the condition:

There is some $(a \asymp b) \in C$ such that neither a nor b are headed by a produced variable.	**I**

Choose one such $a \asymp b$ and continue the case analysis as follows:

1.1 a, b are both rigid and passive expressions. In this case TR is **DC1**, $a = h\,\overline{a}_m$, $b = h\,\overline{b}_m$. The principal type annotation of $\hat{G}$ must include $h^{\overline{\tau}_m \to \tau}\,\overline{a}_m^{\overline{\tau}_m} \asymp h^{\overline{\tau'}_m \to \tau}\,\overline{b}_m^{\overline{\tau'}_m}$. The witness $\mathcal{M}$ must include a type-annotated GORC-proof

$$\begin{aligned}\Pi_0 = (&\Pi \rightsquigarrow h^{(\overline{\tau}_m \to \tau)\theta_t}\,\overline{a}_m\theta_d^{\overline{\tau}_m\theta_t} \to t^{\tau''}\ \&\\ &\Pi' \rightsquigarrow h^{(\overline{\tau'}_m \to \tau)\theta_t}\,\overline{b}_m\theta_d^{\overline{\tau'}_m\theta_t} \to t^{\tau''})+(\mathbf{JN})\end{aligned}$$

where

- $t^{\tau''} = h^{\overline{\tau''}_m \to \tau''}\,\overline{t}_m^{\overline{\tau''}_m}$
- $\overline{\tau''}_m \to \tau'' = (\overline{\tau}_m \to \tau)\theta_t = (\overline{\tau'}_m \to \tau)\theta_t$
- $\Pi = (\ldots\ \&\ \Pi_i \rightsquigarrow a_i\theta_d^{\tau_i\theta_t} \to t_i^{\tau_i''}\ \&\ \ldots)+(\mathbf{DC})$
- $\Pi' = (\ldots\ \&\ \Pi_i' \rightsquigarrow b_i\theta_d^{\tau_i'\theta_t} \to t_i^{\tau_i''}\ \&\ \ldots)+(\mathbf{DC})$

Clearly, **DC1** can be applied to G. Moreover, considering the type-annotated GORC-proofs $\Pi_i'' = (\Pi_i\ \&\ \Pi_i')+(\mathbf{JN}) \rightsquigarrow a_i\theta_d^{\tau_i''} \asymp b_i\theta_d^{\tau_i''}$ $(1 \le i \le m)$ and $\mathcal{M}' =_{def} (\mathcal{M} \setminus \{\{\Pi_0\}\}) \uplus \{\{\Pi_1'', \ldots, \Pi_m''\}\}$, we see that $(R', \theta) \in TASol(\hat{G}')$ with type-annotated witness $\mathcal{M}' \lhd \mathcal{M}$.

1.2 a is a flexible expression, while b is a rigid and passive expression. Due to **I**, the variable X occurring as the head of a is not produced. Moreover, $X\theta_d$ must be a rigid pattern. We distinguish three different subcases:

1.2.1 $a = X\,\overline{a}_k$, $a\theta_d = h\,\overline{t}_m\,\overline{a}_k\theta_d$ is rigid and passive, $b = h\,\overline{s}_m\,\overline{b}_k$, and $s = h\,\overline{s}_m$ is a pattern. In this case TR is **BD**. The principal type annotation

of $\hat{G}$ must include $X^{\overline{\nu}_k \to \nu}\ \overline{a}_k^{\overline{\nu}_k} \asymp (h^{\overline{\mu'}_m \to \overline{\nu'}_k \to \nu}\ \overline{s}_m^{\overline{\mu'}_m})^{\overline{\nu'}_k \to \nu}\ \overline{b}_k^{\overline{\nu'}_k}$. The witness $\mathcal{M}$ must include a type-annotated GORC-proof

$$\Pi_0 = (\Pi \rightsquigarrow X\theta_d^{(\overline{\nu}_k \to \nu)\theta_t}\ \overline{a}_k\theta_d^{\overline{\nu}_k\theta_t} \to t^{\nu''}\ \&\ \Pi' \rightsquigarrow (h^{(\overline{\mu'}_m \to \overline{\nu'}_k \to \nu)\theta_t}\ \overline{s}_m\theta_d^{\overline{\mu'}_m\theta_t})^{(\overline{\nu'}_k \to \nu)\theta_t}\ \overline{b}_k\theta_d^{\overline{\nu'}_k\theta_t} \to t^{\nu''})+(\mathbf{JN})$$

where

- $t^{\nu''} = h^{\overline{\mu''}_m \to \overline{\nu''}_k \to \nu''}\ \overline{t}_m^{\overline{\mu''}_m}\ \overline{u}_k^{\overline{\nu''}_k}$
- (a) $\overline{\nu''}_k \to \nu'' = (\overline{\nu}_k \to \nu)\theta_t = (\overline{\nu'}_k \to \nu)\theta_t$
- $\Pi = (\ldots\ \&\ \Pi_j \rightsquigarrow a_j\theta_d^{\nu''_j} \to u_j^{\nu''_j}\ \&\ \ldots)+(\mathbf{DC})$
- $\Pi' = (\ldots\ \&\ \Pi'_j \rightsquigarrow b_j\theta_d^{\nu''_j} \to u_j^{\nu''_j}\ \&\ \ldots)+(\mathbf{DC})$

In particular, (a) shows that θ_t unifies the types attached to X and s in the principal type annotation of $\hat{G}$. Since θ_t is also solution of $\hat{S}_t$, the dynamic type checking condition of **BD** succeeds and **BD** can actually be applied to transform G into G'. Reasoning as in Lemma 16, $\theta_t = \sigma'_t\theta_t$ and $\theta_d = \rho_d\theta_d$. Taking these identities into account, it is clear that $\mathcal{M}' \lhd \mathcal{M}$ built as indicated below is a type-annotated witness of $(R', \theta) \in \mathit{TASol}(\hat{G}')$: $\mathcal{M}' =_{def} (\mathcal{M} \setminus \{\{\Pi_0\}\}) \uplus \{\{\Pi''_1, \ldots, \Pi''_k\}\}$ where $\Pi''_j = (\Pi_j\ \&\ \Pi'_j)+(\mathbf{JN}) \rightsquigarrow a_j\theta_d^{\nu''_j} \asymp b_j\theta_d^{\nu''_j}$.

1.2.2 $a = X\ \overline{a}_k$, $a\theta_d = h\ \overline{t}_m\ \overline{a}_k\theta_d$ is rigid and passive, $b = h\ \overline{e}_m\ \overline{b}_k$, and $h\ \overline{e}_m$ is not a pattern. In this case TR is **IM**. The principal type annotation of $\hat{G}$ must include $X^{\overline{\nu}_k \to \nu}\ \overline{a}_k^{\overline{\nu}_k} \asymp (h^{\overline{\mu'}_m \to \overline{\nu'}_k \to \nu}\ \overline{e}_m^{\overline{\mu'}_m})^{\overline{\nu'}_k \to \nu}\ \overline{b}_k^{\overline{\nu'}_k}$. The witness $\mathcal{M}$ must include a type-annotated GORC-proof

$$\Pi_0 = (\Pi \rightsquigarrow h^{\overline{\mu''}_m \to (\overline{\nu}_k \to \nu)\theta_t}\ \overline{t}_m^{\overline{\mu''}_m}\ \overline{a}_k\theta_d^{\overline{\nu}_k\theta_t} \to t^{\nu''}\ \& \Pi' \rightsquigarrow h^{(\overline{\mu'}_m \to \overline{\nu'}_k \to \nu)\theta_t}\ \overline{e}_m\theta_d^{\overline{\mu'}_m\theta_t}\ \overline{b}_k\theta_d^{\overline{\nu'}_k\theta_t} \to t^{\nu''})+(\mathbf{JN})$$

where

- $t^{\nu''} = h^{\overline{\mu''}_m \to \overline{\nu''}_k \to \nu''}\ \overline{t}_m^{\overline{\mu''}_m}\ \overline{u}_k^{\overline{\nu''}_k}$
- (a) $\overline{\mu''}_m \to \overline{\nu''}_k \to \nu'' = \overline{\mu''}_m \to (\overline{\nu}_k \to \nu)\theta_t = (\overline{\mu'}_m \to \overline{\nu'}_k \to \nu)\theta_t$
- $\Pi = (\ldots\ \&\ \Delta_i \rightsquigarrow t_i^{\mu''_i} \to t_i^{\mu''_i} \ldots\ \&\ \ldots \Pi_j \rightsquigarrow a_j\theta_d^{\nu''_j} \to u_j^{\nu''_j}\ \&\ \ldots)+(\mathbf{DC})$

- $\Pi' = (\ldots \;\&\; \Delta'_i \rightsquigarrow e_i\theta_d^{\mu''_i} \rightarrow t_i^{\mu''_i} \ldots \;\&\; \ldots \Pi'_j \rightsquigarrow b_j\theta_d^{\nu''_j} \rightarrow u_j^{\nu''_j} \;\&\; \ldots)+(\mathbf{DC})$

In particular, the existence of these proofs implies the condition $X \notin svar(h\ \overline{e}_m)$ needed for the application of **IM**. Moreover, the type $\overline{\mu'}_m \rightarrow \overline{\nu'}_k \rightarrow \nu$ must be an instance of the type $\overline{\tau}_m \rightarrow \tau$ taken as a fresh variant of h's principal type in the formulation of **IM**. Therefore, we can choose some $\rho_t \in TSub$ such that (b) $\overline{\mu'}_m \rightarrow \overline{\nu'}_k \rightarrow \nu = (\overline{\tau}_m \rightarrow \tau)\rho_t$. Consider now the substitution $\theta' = (\theta'_t, \theta'_d)$ such that

- $\theta'_t = \rho_t[tvar(\overline{\tau}_m \rightarrow \tau)]$ and $\theta'_t = \theta_t[\backslash tvar(\overline{\tau}_m \rightarrow \tau)]$.
- $\overline{V}_m\theta'_d = \overline{t}_m$ and $\theta'_d = \theta_d[\backslash \overline{V}_m]$.

Note that $\theta' \approx_{ex} \theta$. Moreover, (a) and (b) imply that θ'_t is a unifier of $\overline{\nu}_k \rightarrow \nu$ (i.e. the type τ' of X in $\hat{T}$) and τ. Moreover, θ'_t is a solution of $\hat{S}_t$, because $\theta'_t = \theta_t[tvar(\hat{S}_t)]$. Therefore, the dynamic type checking condition of **IM** succeeds and **IM** can be used to transform G into G'. Finally, consider $\mathcal{M}' \lhd \mathcal{M}$ built as $\mathcal{M}' =_{def} (\mathcal{M} \setminus \{\{\Pi_0\}\}) \uplus \{\{\Delta''_1, \ldots, \Delta''_m, \Pi''_1, \ldots, \Pi''_k\}\}$ where

- $\Delta''_i = (\Delta_i \;\&\; \Delta'_i)+(\mathbf{JN}) \rightsquigarrow V_i\theta_d^{\mu''_i} \asymp e_i\theta_d^{\mu''_i}$
- $\Pi''_i = (\Pi_i \;\&\; \Pi'_i)+(\mathbf{JN}) \rightsquigarrow a_j\theta_d^{\nu''_j} \asymp b_j\theta_d^{\nu''_j}$

Taking into account the construction of θ' it is easy to check that $\mathcal{M}'$ is a type-annotated witness of $(R', \theta') \in TASol(\hat{G}')$.

1.2.3 $a\theta_d$ is rigid and active. In this case $X\theta_d = f\ \overline{t'}_p$ for some $f \in FS^n$ with $p < n$. Taking $q = n - p > 0$, we can assume $a = X\ \overline{e}_q\ \overline{a}_k$ and $a\theta_d = f\,\overline{t'}_p\,\overline{e}_q\theta_d\,\overline{a}_k\theta_d$. Let us choose **GN** as TR. The principal type annotation of $\hat{G}$ must include (a) $X^{\overline{\nu}_q \rightarrow \overline{\varepsilon}_k \rightarrow \varepsilon}\ \overline{e}_q^{\overline{\nu}_q}\ \overline{a}_k^{\overline{\varepsilon}_k} \asymp b^{\varepsilon}$. In the formulation of **GN**, we find $\tau' = \hat{T}(X)$ and a fresh variant $f :: \overline{\tau}_p \rightarrow \overline{\lambda}_q \rightarrow \tau$ of f's principal type. Therefore, $\tau' = \overline{\nu}_q \rightarrow \overline{\varepsilon}_k \rightarrow \varepsilon$. Moreover, the witness $\mathcal{M}$ must include a type-annotated GORC-proof of (a) using some type-annotated instance of some fresh variant $(f\ \overline{t}_p\ \overline{s}_q \rightarrow r \Leftarrow C' \;\square\; T')$ of a rule of $\mathcal{P}$. Let $\omega = (\omega_t, \omega_d)$ be the substitution which builds the instance of this rule used in the witness. Then, $\mathcal{M}$ will include the following type-annotated GORC-proof

$$\begin{aligned}\Pi_0 = (\Pi \rightsquigarrow f^{\overline{\tau}_p\omega_t \rightarrow (\overline{\nu}_q \rightarrow \overline{\varepsilon}_k \rightarrow \varepsilon)\theta_t}\ \overline{t'}_q^{\overline{\tau}_p\omega_t}\ \overline{e}_q\theta_d^{\overline{\nu}_q\theta_t}\ \overline{a}_k\theta_d^{\overline{\varepsilon}_k\theta_t} \rightarrow t^{\varepsilon\theta_t} \;\&\;\\ \Pi' \rightsquigarrow b\theta_d^{\varepsilon\theta_t} \rightarrow t^{\varepsilon\theta_t})+(\mathbf{JN})\end{aligned}$$

where $t \neq \bot$ is some pattern, and

$$\Pi = (\ldots \,\&\, \Delta_i \rightsquigarrow {t'_i}^{\tau_i\omega_t} \to t_i\omega_d^{\tau_i\omega_t} \ldots \,\&\, \ldots \Pi_j \rightsquigarrow e_j\theta_d^{\nu_j\theta_t} \to s_j\omega_d^{\lambda_j\omega_t} \,\&\, \ldots \,\&\, \mathcal{M}_0 \rightsquigarrow C'\omega_d^{\mathtt{bool}} \,\&\, \Pi'' \rightsquigarrow r\omega_d^{\tau\omega_t}\,\overline{a}_k\theta_d^{\overline{\varepsilon}_k\theta_t} \to t^{\varepsilon\theta_t})+(\mathbf{OR})$$

Since the previous proofs are type-annotated, the following type identities must hold: (b) $\overline{\lambda}_q\omega_t = \overline{\nu}_q\theta_t$, (c) $\tau\omega_t = (\overline{\varepsilon}_k \to \varepsilon)\theta_t$. Consider now the substitution $\theta' = (\theta'_t, \theta'_d)$ such that

- $\theta'_t = \omega_t[dom(\omega_t)]$ and $\theta'_t = \theta_t[\backslash dom(\omega_t)]$.
- $\theta'_d = \omega_d[dom(\omega_d)]$ and $\theta'_d = \theta_d[\backslash dom(\omega_d)]$.

By construction, $\theta' \approx_{ex} \theta$. Because of (b) and (c) θ'_t is a unifier of $\overline{\nu}_q \to \overline{\varepsilon}_k \to \varepsilon$ and $\overline{\lambda}_q \to \tau$. Moreover, $\theta'_t = \theta_t[tvar(\hat{S}_t)]$ implies $\theta'_t \in TSol(\hat{S}_t)$. Therefore, the dynamic type checking condition of **GN** succeeds and **GN** can be used to transform G into G'. To finish this case, we need a type-annotated $\mathcal{M}' \lhd \mathcal{M}$ witnessing $(R', \theta') \in TASol(\hat{G}')$. Taking into account the construction of θ', it is easy to check that the following $\mathcal{M}'$ does the job: $\mathcal{M}' =_{def} (\mathcal{M} \setminus \{\{\Pi_0\}\}) \uplus \mathcal{M}_0 \uplus \{\{\Pi_1, \ldots, \Pi_q, \Pi'''\}\}$ where $\Pi''' = (\Pi'' \,\&\, \Pi')$ $+(\mathbf{JN}) \rightsquigarrow r\omega_d^{\tau\omega_t}\,\overline{a}_k\theta_d^{\overline{\varepsilon}_k\theta_t} \asymp b\theta_d^{\varepsilon\theta_t}$.

1.3 Both a and b are flexible expressions. In this case we distinguish four subcases.

1.3.1 $a = X\,\overline{a}_p$ and $b = X\,\overline{b}_q$ where $a\theta_d$ (and thus $b\theta_d$) is a rigid and passive. This case can be handled by taking **ID** as TR.

1.3.2 a and b are two different variables, say X and Y. This case can be handled by choosing TR as **BD**, with $k = 0$ and $s = Y$. In this simple situation, **BD** needs no dynamic type checking.

1.3.3 $a = X\,\overline{a}_p$ and $b = Y\,\overline{b}_q$ where X and Y are different variables and $a\theta_d$, $b\theta_d$ are both rigid and passive. Without loss of generality, we can assume $p \geq q$. The witness must include a GORC proof of $(a \asymp b)\theta_d$. This is possible only if $X\theta_d$ and $Y\theta_d$ are patterns of the forms $h\,\overline{t}_{m-p}$ and $h\,\overline{s}_{m-q}$, respectively. Moreover, since $p \geq q$ implies $m-p \leq m-q$, the sequence $\overline{t}_{m-p}$ must coincide with the $m-p$ first patterns in the sequence $\overline{s}_{m-q}$. Moreover, the GORC proof of $(a \asymp b)\theta_d$ must end with a **JN** inference with two premises of the form $a\theta_d \to h\,\overline{t}_{m-p}\,\overline{u}_p$ and $b\theta_d \to h\,\overline{s}_{m-q}\,\overline{v}_q$ (where $h\,\overline{t}_{m-p}\,\overline{u}_p$ and $h\,\overline{s}_{m-q}\,\overline{v}_q$ are identical patterns), each of them proved by a **DC** inference. Taking all

these ideas into account, this case can be completed by choosing **GD** as TR and exploiting the type information included in $\mathcal{M}$. We omit the detailed reasoning, which would be somewhat similar to that of case **1.2.2**.

1.3.4 $a\theta_d$ is rigid and active. Let X be the variable occurring as the head of a. As in case **1.2.3**, $X\theta_d$ must be a pattern $f\ \overline{t'}_p$ for some $f \in FS^n$ with $p < n$. Taking $q = n - p > 0$, we can assume $a = X\ \overline{e}_q\ \overline{a}_k$ and $a\theta_d = f\ \overline{t'}_p\ \overline{e}_q\theta_d\ \overline{a}_k\theta_d$. Note that the assumption "b rigid and passive" in case **1.2.3** has been actually not used in our proof for this case. Therefore, the same proof can be reused to complete the present case, choosing **GN** as TR.

1.4 a is rigid and active expression. In this case, we can assume $a = f\ \overline{e}_n\ \overline{a}_k$ for some $f \in FS^n$ with $k \geq 0$. The GORC proof of $(a \asymp b)\theta_d$ included in $\mathcal{M}$ must involved an **OR**-inference using some type-annotated instance of a freshly renamed program rule $Rl : (f\ \overline{t}_n \to r \Leftarrow C' \ \square\ T') \in_{var} \mathcal{P}$, build with a well-typed substitution $\omega = (\omega_t, \omega_d)$. More precisely, the principal type annotation of $\hat{G}$ must include $f^{\overline{\nu}_n \to \overline{\varepsilon}_k \to \varepsilon}\ \overline{e}_n^{\overline{\nu}_n}\ \overline{a}_k^{\overline{\varepsilon}_k} \asymp b^{\varepsilon}$ while $\mathcal{M}$ must include a type-annotated GORC-proof

$$\Pi_0 = (\Pi \rightsquigarrow f^{(\overline{\nu}_n \to \overline{\varepsilon}_k \to \varepsilon)\theta_t}\ \overline{e}_n\theta_d^{\overline{\nu}_n\theta_t}\ \overline{a}_k\theta_d^{\overline{\varepsilon}_k\theta_t} \to t^{\varepsilon\theta_t}\ \&\ \Pi' \rightsquigarrow b\theta_d^{\varepsilon\theta_t} \to t^{\varepsilon\theta_t}) + (\mathbf{JN})$$

where $t \neq \bot$ is some pattern, and

$$\Pi = (\ldots\ \&\ \Pi_i \rightsquigarrow e_i\theta_d^{\nu_i\theta_t} \to t_i\omega_d^{\tau_i\omega_t}\ \&\ \ldots\ \&\ \mathcal{M}_0 \rightsquigarrow C'\omega_d^{\mathtt{bool}}\ \&\ \Pi'' \rightsquigarrow r\omega_d^{\tau\omega_t}\ \overline{a}_k\theta_d^{\overline{\varepsilon}_k\theta_t} \to t^{\varepsilon\theta_t}) + (\mathbf{OR})$$

Note that Π must be built using the ω instance of the principal type annotation of Rl, which we assume to be $(f^{\overline{\tau}_n \to \tau}\ \overline{t}_n^{\overline{\tau}_n} \to r^{\tau} \Leftarrow C'^{\mathtt{bool}})$. Since Π_0 is a type-annotated proof, the type identity $(\overline{\tau}_n \to \tau)\omega_t = (\overline{\nu}_n \to \overline{\varepsilon}_k \to \varepsilon)\theta_t$ must hold. The rest of this case can be completed by choosing **NR1** as TR. The proof details are similar to those of case **1.2.3**, but easier. In particular, no dynamic type checking is needed.

2 Now we assume that G satisfies the condition $\neg\mathbf{I} \wedge \mathbf{II}$, where

> There is some $(e \to t) \in P$ such that $t \notin DVar$. **II**

Note that $\neg\mathbf{I}$ can be rewritten as:

> For all $(a \asymp b) \in C$, either a or b is headed by a produced variable. $\neg\mathbf{I}$

Choose any $(e \to t) \in P$ such that t is not a variable, and continue the case analysis as follows:

2.1 e is a rigid and passive expression. Note that e and t must have the forms $h\,\overline{e}_n$ and $h\,\overline{t}_n$, respectively. Otherwise, $(e \to t)\theta_d$ could have no GORC proof. The rest of this case can be completed by taking **DC2** as TR, by a reasoning similar to that of case **1.1**.

2.2 e is a rigid and active expression. This case can be proved using **NR2** as TR. Note that **NR2** can be applied because t is not a variable. The proof details are similar to those of case **1.4**.

2.3 e is flexible, but $e\theta_d$ is rigid and passive. In this case we can assume $(e \to t) = (X\,\overline{e}_k \to h\,\overline{t}_m\,\overline{s}_k)$ and $X\theta_d = h\,\overline{t'}_m$. Since θ_d is a solution of $(e \to t)$, we can also assume $h\,\overline{t'}_m \sqsupseteq h\,\overline{t}_m\theta_d$. To continue our reasoning, we choose **OB** as TR. The principal type annotation of $\hat{G}$ must include

$$X^{\overline{\nu}_k \to \nu}\,\overline{e}_k^{\overline{\nu}_k} \to h^{\overline{\mu}_m \to \overline{\nu'}_k \to \nu}\,\overline{t}_m^{\overline{\mu}_m}\,\overline{s}_k^{\overline{\nu'}_k}$$

and $\mathcal{M}$ must include a type-annotated GORC-proof

$$\Pi_0 = (\ldots \,\&\, \Delta_i \rightsquigarrow t_i'^{\mu_i'} \to t_i\theta_d^{\mu_i\theta_t} \ldots \,\&\, \ldots \Pi_j \rightsquigarrow e_j\theta_d^{\nu_j\theta_t} \to s_j\theta_d^{\nu_j'\theta_t} \,\&\, \ldots)$$
$$+(\mathbf{DC}) \rightsquigarrow (h^{\overline{\mu'}_m \to (\overline{\nu}_k \to \nu)\theta_t}\,\overline{t'}_m^{\overline{\mu'}_m}\,\overline{e}_k\theta_d^{\overline{\nu}_k\theta_t} \to h^{(\overline{\mu}_m \to \overline{\nu'}_k \to \nu)\theta_t}\,\overline{t}_m\theta_d^{\overline{\mu}_m\theta_t}\,\overline{s}_k\theta_d^{\overline{\nu'}_k\theta_t})$$

In the formulation of **OB** we find $\tau' = \hat{T}(X)$ and $h :: \overline{\tau}_m \to \tau$ as fresh variant of h's principal type. Since the type annotations in Π_0 must be consistent, the following type identities must hold for some $\rho_t \in \mathit{TSub}$:

(a) $\overline{\mu'}_m \to (\overline{\nu}_k \to \nu)\theta_t = (\overline{\mu}_m \to \overline{\nu'}_k \to \nu)\theta_t$.

(b) $\overline{\mu}_m \to \overline{\nu'}_k \to \nu = (\overline{\tau}_m \to \tau)\rho_t$.

Consider $\theta_t' \in \mathit{TSub}$ defined so that $\theta_t' = \rho_t[\mathit{tvar}(\overline{\tau}_m \to \tau)]$ and $\theta_t' = \theta_t[\backslash \mathit{tvar}(\overline{\tau}_m \to \tau)]$. Because of (a) and (b), θ_t' is a unifier of $\tau' = \hat{T}(X) = \overline{\nu}_k \to \nu$ and τ. Moreover, $\theta_t' \in \mathit{Sol}(\hat{S}_t)$, because $\theta_t' = \theta_t[\mathit{tvar}(\hat{S}_t)]$. Hence, θ_t' is a solution of $(\hat{S}_t, \tau \approx \tau')$, the dynamic type checking condition of **OB** succeeds, and **OB** can be applied to transform G into G'.

We still have to build $\theta' \approx_{ex} \theta$ such that $(R', \theta') \in \mathit{TASol}(\hat{G}')$ with some type-annotated witness $\mathcal{M}' \lhd \mathcal{M}$. In particular, for θ_d' to be a solution of $X \approx h\,\overline{t}_m$, the identity $X\theta_d' = h\,\overline{t}_m\theta_d'$ should hold. For θ_d, however, we

only know $X\theta_d = h\,\overline{t'}_m \sqsupseteq h\,\overline{t}_m\theta_d$. Therefore, we are going to define θ'_d as a modification of θ_d acting differently over $var(h\,\overline{t}_m)$. Thanks to the condition **EX** of admissible goals, $var(h\,\overline{t}_m)$ consists of existential variables, which ensures $\theta'_d \approx_{ex} \theta_d$.

To build θ'_d, assume $var(h\overline{t}_m) = \{Y_1, \ldots, Y_p\}$. Since $h\,\overline{t}_m$ is linear, each Y_j has exactly one occurrence at one position in $h\,\overline{t}_m$. Let u_j be the subpattern which occurs at the same position in $h\,\overline{t}_m\theta'_d$ and let u'_j be the subpattern which occurs at the same position in $h\,\overline{t'}_m$. Note that $h\,\overline{t'}_m \sqsupseteq h\,\overline{t}_m\theta_d$ implies $u'_j \sqsupseteq u_j$ for all $1 \leq j \leq p$. Define $\theta'_d \in DSub_\perp$ such that $Y_j\theta'_d = u'_j$, $1 \leq j \leq p$ and $\theta'_d = \theta_d[\backslash\{Y_1, \ldots, Y_p\}]$. In particular, $X\theta'_d = X\theta_d = h\,\overline{t'}_m$, because $X \notin var(h\,\overline{t}_m)$ due to the **NC** condition of admissible goals. Hence, $X\theta'_d = h\,\overline{t'}_m = h\,\overline{t}_m\theta'_d$.

Finally, regarding the witness $\mathcal{M}' \lhd \mathcal{M}$ of $(R, \theta') \in TASol(\hat{G}')$, let us consider $\mathcal{M}'_0 =_{def} (\mathcal{M}\backslash\{\{\Pi_0\}\}) \uplus \{\{\Pi_1, \ldots, \Pi_k\}\}$ which is easily seen to serve as a type-annotated witness for the principal type annotation of $\hat{G}'$ affected by (θ'_t, θ_d). By applying Lemma 8 (item 3), $\mathcal{M}'_0$ can be transformed into another witness $\mathcal{M}'$ with the same structure for the principal type annotation of $\hat{G}'$ affected by (θ'_t, θ'_d). Note that Lemma 8 says nothing about type annotations, but this is not a problem, because $\mathcal{M}'$ and $\mathcal{M}'_0$ are the same, except that certain occurrences of u_j in $\mathcal{M}'_0$ are replaced by occurrences of u'_j in $\mathcal{M}'$, which can inherit the type annotations of u'_j as a subpattern of $h\,\overline{t'}_m$.

2.4 e is flexible, but $e\theta_d$ is rigid and active. This case can be proved by using **OGN** as TR. Note that **OGN** can be applied because t is not a variable. The proof details are similar to those of case **1.2.3**.

3 Now we assume that G satisfies the condition $\neg\mathbf{I} \wedge \neg\mathbf{II} \wedge \mathbf{III}$, where $\neg\mathbf{II}$ can be rewritten as:

$$P \text{ includes only statements } (e \to X) \text{ with } X \in DVar \qquad \neg\mathbf{II}$$

and

$$\text{There is some } (t \to X) \in P, \text{ such that } t \text{ is a pattern.} \qquad \mathbf{III}$$

This case is easy to prove taking **IB** as TR.

4 Assume now that G satisfies the condition $\neg\mathbf{I} \wedge \neg\mathbf{II} \wedge \neg\mathbf{III} \wedge \mathbf{IV}$, where

$$C \neq \emptyset. \qquad \mathbf{IV}$$

Because of **IV** and $\neg$**I**, we can choose some $(a \asymp b) \in C$ such that a is headed by a produced variable Y. Since Y is produced and $\neg$**II** $\wedge$ $\neg$**III** hold, $(e \to Y) \in P$ for some expression e which is not a pattern. Moreover, Y is demanded because it occurs as the head of a in $(a \asymp b) \in C$. We distinguish four subcases according to the form of e and $e\theta_d$.

4.1 e is rigid and passive. This case can be proved by taking **IIM** as TR. The reasoning is similar to that of case **1.2.2** (with $k = 0$), but simpler. In particular, no dynamic type checking is needed.

4.2 e is rigid and active. In this case we can choose **NR2** as TR. Note that **NR2** can be applied because Y is demanded. The proof details are similar to case **1.4**.

4.3 e is flexible, but $e\theta_d$ is rigid and passive. Since e is not a pattern, we can assume $e = X\,\overline{e}_q$ with $q > 0$. Since θ_d is solution of $e\theta_d$, $X\theta_d$ and $Y\theta_d$ must be patterns of the form $h\,\overline{t'}_p$ and $h\,\overline{t'}_p\overline{s}_q$, respectively, and $\mathcal{M}$ must include a GORC proof of $h\,\overline{t'}_p\overline{e}_q\theta_d \to h\,\overline{t'}_p\overline{s}_q$ ending with a **DC** step. The rest of this case can be completed by taking **OGD** as TR.

4.4 e is flexible, but $e\theta_d$ is rigid and active. This case quite similar to case **2.4**. It can be proved by using **OGN** as TR. Note that **OGN** can be applied because Y is a demanded variable.

5 Finally, assume that G satisfies the condition $\neg$**I** $\wedge$ $\neg$**II** $\wedge$ $\neg$**III** $\wedge$ $\neg$**IV**. Since G is not solved and **IV** is false, $C = \emptyset$ and $P \neq \emptyset$. Due to the acyclicity condition **NC** of admissible goals, P must contain some statement $(e \to t)$ such that $var(t) \cap var(e') = \emptyset$ for all $(e' \to t') \in P$. Due to $\neg$**II**, t must be a single variable X. Because of $C = \emptyset$ and the linearity condition **LN** of admissible goals, X has exactly one occurrence in G. Then **EL** can be applied to transform G into G', and a type-annotated witness $\mathcal{M}' \lhd \mathcal{M}$ for $(R, \theta') \in TASol(\hat{G}')$ can be obtained from $\mathcal{M}$ by removing the type-annotated proof of $(e \to X)\theta_d$. $\Box$