TLP **17** (5–6): 992–1009, 2017. © Cambridge University Press 2017
doi:10.1017/S1471068417000230 First published online 22 August 2017

*Constraint handling rules with binders, patterns and generic quantification**

ALEJANDRO SERRANO and JURRIAAN HAGE
Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands
(*e-mails:* A.SerranoMena@uu.nl, J.Hage@uu.nl)

submitted 2 May 2017; revised 20 June 2017; accepted 4 July 2017

Abstract

Constraint handling rules provide descriptions for constraint solvers. However, they fall short when those constraints specify some binding structure, like higher-rank types in a constraint-based type inference algorithm. In this paper, the term syntax of constraints is replaced by λ-tree syntax, in which binding is explicit, and a new ∇ generic quantifier is introduced, which is used to create new fresh constants.

KEYWORDS: Constraint Handling Rules, pattern unification, generic quantification

1 Introduction

Constraint handling rules (Frühwirth 2009) – usually shortened to CHRs – provide a language to describe constraint solvers. The great body of work related to CHRs, summarized in the papers by Frühwirth (1998) and Sneyers *et al.* (2010), makes them a good choice to describe algorithms where constraint rewriting is involved.

Type inference for functional languages is one of the areas in which CHRs have been applied successfully. CHRs have been used to improve type error reporting (Stuckey *et al.* 2006; Wazny 2006; Serrano and Hage 2016), describe and extend the type class machinery in Haskell (Dijkstra *et al.* 2007; Sulzmann *et al.* 2007) and generalize the shape of algebraic data types (Sulzmann *et al.* 2006). However, CHRs are not enough to describe in a concise form,[1] the solving needed for some advanced type system features.

In particular, consider parametric polymorphism. In languages based on the Hindley–Damas–Milner typing discipline, such as Haskell and ML, we find a stratification of types. Expressions are assigned a simple type, whereas declarations are given a *type scheme*, which quantifies over a set of variables. For example, the identity function *id* types with the following type scheme: $\forall a.\, a \to a$.

* This work was supported by the Netherlands Organisation for Scientific Research (NWO) project on "DOMain Specific Type Error Diagnosis (DOMSTED)" (612.001.213).

[1] Since CHRs are Turing-complete (Sneyers *et al.* 2009), in theory any algorithm can be described. This does not mean that the description is convenient, though.

https://doi.org/10.1017/S1471068417000230 Published online by Cambridge University Press

At each point where the *id* function is used as an expression, we cannot directly use this type scheme, since expressions must be assigned a type. Thus, we are forced to remove the quantification by *instantiating* a. In other words, every occurrence of *id* is assigned a type $\alpha \to \alpha$ for a fresh variable α. Solving is responsible for finding a type for that variable, or deciding to quantify over it.

One of the beauties of Hindley–Damas–Milner is that we can instantiate the type schemes in a binding block at once before we start solving. Instantiating variables during constraint gathering takes us quite far: the type system of Haskell as of GHC 7, including type classes, type families and GADTs, has been described in this fashion (Vytiniotis *et al.* 2011). However, when type application (Eisenberg *et al.* 2016) or higher-rank types (Peyton Jones *et al.* 2007) enter the picture, we need to *delay* instantiation.

As a result, whereas before each use of *id* in the source code would lead to the type $\alpha \to \alpha$ for a fresh variable α, now we simply assign it a fresh type β and recall the instantiation relation by means of a constraint $\forall\, a.a \to a \leqslant \beta$. The ball of finding a good type assignment for β is now in the court of the constraint solver, which we describe using CHRs. A rule instantiating polymorphic types looks similar to

$$\mathit{forall}\ (X, T) \leqslant S \iff T' = \mathit{inst}\ (X, T) \mid T' = S$$

intutively, this rule states that every time it finds an $R \leqslant S$ constraint where the left-hand side is polymorphic – $\mathit{forall}\ (X, T)$ – it should replace it by a type equality $T' = S$, where T' is derived from R by instantiating X. There are some problems with this rule:

(1) There is no guarantee that variables are correctly bound in polymorphic types. We need to rely on the invariant that we are allowed to use variable X only inside T.
(2) How to adequately represent type variables within CHRs is not yet well understood (Csorba *et al.* 2012). The simplest solution, using a unique number per variable, does not work well with binding structures.
(3) The *inst* function has to be defined *externally* to the constraint solver. Generation of fresh variables needs some extra state not reflected directly in the rules. Furthermore, instantiation is tricky due to variable capture.

In this paper, we propose an extension of CHR, which we call CHR$^{\nabla}$, in which the binding structure of a term is explicit and represented by λ-abstractions: $\mathit{forall}\ (\lambda X.\, T)$. This syntax for binders is called λ-tree syntax (Miller 2000). The previous constraint transformation is now written as

$$(1) \qquad \mathit{forall}\ (T) \leqslant S \iff \exists V.\, T\ V \leqslant S$$

In the body, we create a new variable V by means of the $\exists$ operator, and then we replace X with V in the body of T. The system ensures that no variable is incorrectly captured.

We can go a step further and consider the shape of the rule whenever a polymorphic type is present to the right of the $\leqslant$ constraint. The usual definition

https://doi.org/10.1017/S1471068417000230 Published online by Cambridge University Press

of instantiation says that $\sigma \leqslant \forall a.\tau$ if and only if $\sigma \leqslant \tau[a \mapsto \rho]$ for any choice of ρ. One way to prove this fact is by introducing a *nominal constant* $\hat{a}$ standing for a, that is, a constant distinct from any other term in the language, and proving $\sigma \leqslant \tau[a \mapsto \hat{a}]$. Nominal constraints are also referred to as rigid or Skolem variables (we use the three names interchangeably throughout the paper). Our extension to CHRs includes a ∇ operator, inspired by the logic of Abella (Tiu 2006; Baelde *et al.* 2014), which introduces a new nominal constant.

$$T \leqslant \mathit{forall}\ (S) \iff \nabla A.\, T \leqslant S\ A \tag{2}$$

Note that in order to have a sound algorithm, rule (1) must always be applied before rule (2). User-definable rule priorities (Koninck *et al.* 2007) add support for preferences in CHRs. Priorities, on the other hand, are orthogonal to our extensions to CHR, and thus we concern ourselves only with "classical" rules.

Contributions: Specifically, our contributions in this paper are as follows:

- Extending CHR matching from ground terms to λ-tree terms via L_λ-unification.
- Incorporating the notion of nominal constants and the ∇ operator to generate fresh instances of these constants.
- Providing techniques to deal with confluence and termination in this new scenario.

As an example of the power of the framework, we showcase an extension to the Haskell language to provide simple higher-rank types.

The integration of λ-tree syntax with ∇ was already present in the Abella theorem prover (Baelde *et al.* 2014), the integration with CHRs is entirely novel.

2 Preliminaries

In this section, we give a brief introduction to each framework involved in our work.

2.1 Constraint handling rules

The language of CHRs has three kinds of rules:

$$\begin{array}{lcl} H^r & \iff & G \mid B \quad \text{simplification} \\ H^k & \Longrightarrow & G \mid B \quad \text{propagation} \\ H^k \setminus H^r & \iff & G \mid B \quad \text{simpagation} \end{array}$$

In each case, H^k, H^r and B are sets of constraints, called the heads and the body, respectively. We use $\top$ to represent an empty set of constraints (reminiscent of "true"). In order for a rule to be applied, some constraints from the current set must match the heads, and the guard G must be satisfied. Rewriting depends on the kind of rule: with simplification rules the constraints H^r are replaced by B, in propagation rules the constraints B are added to the set but the constraints H^k are kept. Simpagation rules are a generalization of both: H^k constraints are kept and H^r are removed. In fact, we can view any CHR as a simpagation rule where the heads might be empty.

https://doi.org/10.1017/S1471068417000230 Published online by Cambridge University Press

CHRs are applied non-deterministically. For a given initial constraint set, many different sequences of applications of rules are usually possible. Confluence, that is, the fact that the outcome of the process does not depend on the order in which rules are applied, must be proven externally by the author of the CHRs.

Type classes in Haskell are a prime example of what can be described using the CHR language (Sulzmann *et al.* 2007). A declaration of the following type classes and instances

class $Eq\ a$ **where**...	**class** $Eq\ a \Rightarrow Ord\ a$ **where**...
instance $Eq\ Int$ **where**...	**instance** $Ord\ Int$ **where**...
instance $Eq\ a \Rightarrow Eq\ [a]$ **where**..	**instance** $Ord\ a \Rightarrow Ord\ [a]$ **where**...

gives rise to the following set of rules:

$$\begin{array}{ll} & Ord\ A \implies Eq\ A \\ Eq\ Int \iff \top & Ord\ Int \iff \top \\ Eq\ [A] \iff Eq\ A & Ord\ [A] \iff Ord\ A \end{array}$$

Using these rules, we have different ways to get from $Ord\ [a]$ to $Eq\ a$, illustrating the non-deterministic nature of CHRs. On the one hand, $Ord\ [a]$ can be simplified to $Ord\ a$, which then generates a constraint $Eq\ a$. On the other hand, the generation of $Eq\ [a]$ might take place first, and only then will it be simplified to $Eq\ a$. The restrictions that the Haskell language imposes on type classes ensure that both paths are equivalent. When translated to CHRs, the resulting rules form a confluent set.

2.2 λ-tree syntax, L_λ-unification and β_0-reduction

Several approaches exist for the representation of binding inside a language in a both convenient and efficient way, including de Bruijn indices (de Bruijn 1972), locally nameless representation (Charguéraud 2012) and extensions dealing with hygiene in macros and different namespaces. In this paper, we use λ-tree syntax (Miller 2000), which is closely related to higher-order abstract syntax (Pfenning and Elliott 1988).

Consider λ-terms, which extend a base language of terms with a binding operator λ, variables x and an application form $T_1\ T_2$. The equality relation between λ-terms includes not only syntactic equality but also the following three rules:

$$\begin{array}{rcll} \lambda x.\, B & = & \lambda y.\, B[x \mapsto y] \quad \text{if } y \text{ not free in } B & (\alpha) \\ (\lambda x.\, B)\, E & = & B[x \mapsto E] \text{ if } E \text{ does not contain } x & (\beta) \\ \lambda x.\, F\ x & = & F & (\eta) \end{array}$$

During solving, the CHR engine needs to check whether any subset of the active constraints matches a rule. In order to do so, it matches the constraints with the patterns appearing in the head of the rule. In the setting of normal CHRs using first-order terms, this check is unification. But now we have λ-terms, leading to higher-order unification (Huet 1975). Full higher-order unification has several drawbacks, including undecidability. Miller (2000) argues for a weaker matching procedure, but which guarantees decidability, finds most general unifiers and runs in linear time (Qian 1993). This procedure is called L_λ-unification, or (higher-order) pattern unification.

https://doi.org/10.1017/S1471068417000230 Published online by Cambridge University Press

The main restriction in pattern unification is that an application in which the head is a metavariable must be done to distinct bound variables. Using a variable repeatedly is not allowed, so matching with $\lambda x.\lambda y.F\ x\ y$ is OK, but $\lambda x.F\ x\ x$ is not. Using a term that is not a variable as an argument, as in $\lambda x.F\ (G\ x)$, is not allowed either. The only exception is the η-expansion of variables: The term $\lambda x.F\ (\lambda z.x\ x)$ is allowed, since it is equivalent to $\lambda x.F\ x$.

The theory of unification of patterns is actually a theory of equality of λ-terms with the above restrictions. In such a scenario, the full power of β-reduction is not needed, but just a restricted version for variables:

$$(\lambda x.B)\,y \quad = \quad B[x \mapsto y] \qquad (\text{rule } \beta_0)$$

The reader may be worried about patterns being overly restrictive for our purposes. Miller (1991) argues that for practical purposes, pattern unification is enough, citing developments in the Isabelle theorem prover and the λProlog logic system. Later developments, such as the Abella theorem prover, also make use of L_λ-unification. Thus, patterns seem to be a sweet spot to base a language on.

2.3 *The generic quantifier* ∇

As Miller and Tiu discuss in their paper (Miller and Tiu 2005), there are two ways to prove a universally quantified proposition $\forall x.F$. The first one is proving $F[x \mapsto T]$ for every closed term T. Usually the set of terms is inductively defined, in which case *induction* can be used to reduce the number of cases. Another possibility is proving $F[x \mapsto c]$ for a completely new nominal constant c, which shall remain fresh during the whole proof.

These two notions are not completely interchangeable: $\forall x\,y.P(x,y) \implies \forall z.P(z,z)$ holds for the first approach, since you can instantiate x and y in the first universal with the same value z. But $\forall x\,y.P(x,y) \implies \forall z.P(z,z)$ does *not* hold in general with the second reading: We instantiate the antecedent with two different constants, leading to $P(a,b)$. But now we cannot make a equal to b – which we need to prove $\forall z.P(z,z)$ – all we know about those constants is indeed that they are different!

In order to distinguish these different mechanics, Miller and Tiu (2005) introduce a new *generic quantifier* ∇ to account for the second nature. During proof search, ∇ introduces a new scoped constant, different from any other such constant in the proof. The resulting logic was extended (Tiu 2006) to account for some desirable properties of ∇, such as $\nabla x.B$ being equivalent to B whenever x is not free in B. The resulting logic, LG^ω, forms the basis of our work. Formally, the rules governing this new universal quantifier ∇ are

$$\frac{\Gamma, B[x \mapsto \mathsf{a}] \vdash C \qquad a \notin \mathsf{supp}(B)}{\Gamma, \nabla x.B \vdash C}\ \nabla_{\mathrm{L}}$$

$$\frac{\Gamma \vdash C[x \mapsto a] \qquad a \notin \mathsf{supp}(C)}{\Gamma \vdash \nabla x.C}\ \nabla_{\mathrm{R}}$$

$$\frac{\pi(B) \equiv \pi'(B') \qquad \pi, \pi' \text{ permutations of constants}}{\Gamma, B \vdash B'}\ \mathrm{ID}\pi$$

https://doi.org/10.1017/S1471068417000230 Published online by Cambridge University Press

Variables	$\mathfrak{V}$	$\ni$	$X, Y, Q, R, T, V \ldots$	
Nominal constants	$\mathfrak{K}$	$\ni$	$\mathtt{a}, \mathtt{b}, \ldots$	
Constraints / patterns	C	$::=$	$c(T_1, \ldots, T_n)$	$c \in \mathfrak{C}$
Terms	T	$::=$	$X \mid \mathtt{a}$	
		$\mid$	$f(T_1, \ldots, T_n)$	$f \in \mathfrak{F}$
		$\mid$	$\lambda X.T$	
		$\mid$	$T_1\ T_2$	

Fig. 1. Syntax of constraints and terms.

In the rules, the *support* of a formula B, $\mathsf{supp}(B)$, is defined as the set of scoped constants in B. The reader can see that left- and right-introduction rules for ∇ work in the same way: ∇ is indeed a self-dual quantifier.

Having only these two rules is not enough to prove some of the theorems we would like to hold; in particular, $\nabla x.B(x) \vdash \nabla y.B(y)$ is not true. The reason is that each ∇ introduces its own fresh constant, which are guaranteed to be distinct. More of these "non-examples" can be found in Figure 4 of the paper by Miller and Tiu (2005). The solution is to allow a permutation of constants in both the antecedent and the consequent, as shown in rule $\textsc{id}\pi$. In our case, assume that after the introduction of constants, we have to prove $B(\mathtt{a}) \vdash B(\mathtt{b})$ for constants $\mathtt{a}$ and $\mathtt{b}$. By applying an identity permutation on the left and the permutation $[\mathtt{a} \mapsto \mathtt{b}, \mathtt{b} \mapsto \mathtt{a}]$ on the right, we obtain syntactically equal formulas.

The previous rules make ∇ commute with the $\vee$, $\wedge$ and $\supset$ connectives. Swapping of constants respects provability: $\nabla x.\nabla y.B(x, y) \equiv \nabla y.\nabla x.B(x, y)$, as witnessed by one application of $\textsc{id}\pi$. Finally, we have that $\forall x.B(x) \vdash \nabla x.B(x)$ and $\nabla x.B(x) \vdash \exists x.B(x)$.

The problem that led to the creation of ∇ was to reason about an object logic in a different meta-logic (Tiu 2006). This idea fits our problem: we want to reason about constraints with universal quantification, our object logic, at a higher level, namely CHRs. As we shall see in Section 3, each appearance of a ∇ in a rule leads to the introduction of a new scoped constant.

3 Putting all together: CHR$^{\nabla}$

In this section, we introduce our extensions to the CHR machinery needed to cope with rules such as those in Section 1. A simple implementation of type inference for higher-rank types is presented as an example of its use.

Syntax: In the syntax of CHR$^{\nabla}$, we use four sets of objects. These sets must be disjoint, except for constraint constructors and term constructors, which may overlap.

- A set of *constraint* constructors $\mathfrak{C}$ annotated with arity.
- A set of *term* constructors $\mathfrak{F}$ annotated with their arity. We denote both types of constructors by lowercase letters such as $c, f, g, \ldots$
- An infinite set of *term variables* $\mathfrak{V}$, which we denote by uppercase letters $X, Y, \ldots$
- An infinite set of *nominal constants* $\mathfrak{K}$, which we denote by teletype letters $\mathtt{a}, \mathtt{b}, \ldots$

https://doi.org/10.1017/S1471068417000230 Published online by Cambridge University Press

Using these sets, we build up both constraints and terms, as given in Figure 1. Note that at the constraint level, abstraction and application are not permitted; this richer syntactic structure is only available to terms.

In CHR$^{\nabla}$, the syntax of terms and patterns coincide; we use the second term to emphasize the role of a term being part of a rule. In the following, we often use C to refer to both single constraints or sets of them; the context is enough to distinguish the intended meaning.

One of our goals in this paper is to allow the more powerful L_{λ}-unification to be used instead of plain term unification. In order to do so, we need to restrict the shape of constraints which may appear, as described in Section 2.2.

Definition 1 (*Well-defined patterns and constraints*)
We say that a pattern P is *well-defined* if and only if free variables appear only applied to distinct variables or η-equivalent versions of these.
We say that a constraint C is *well-defined* if and only if each of its terms is a well-defined pattern and does not contain any nominal constant.

Finally, we can describe the syntax of a rule in CHR$^{\nabla}$:

$$H^k \setminus H^r \iff G \mid \nabla X_1 \ldots X_n. \exists Y_1 \ldots Y_m. B$$

where H^k, H^r are sets of well-defined constraints, and all free variables in B come from either the free variables in H^k and H^r or from $\{X_1, \ldots, X_n, Y_1, \ldots, Y_m\}$. Both the set of universally quantified and of existentially quantified variables may be empty. Note that all variables in a set of constraints must come from either the initial set or introduced by explicit quantification. In contrast, "classical" CHRs quantify variables implicitly.

We define simplification and propagation rules as a restriction of the main kind of rule with empty H^k and H^r, respectively, as shown in Section 2.1.

Declarative semantics: There are different ways to interpret a set of CHRs. In other words, we can attach different *semantics* to them. The *declarative* semantics (Frühwirth 1998) maps each rule to a logic formula. First, let us consider the declarative semantics of a rule without any of our extensions, that is, $H^k \setminus H^r \iff G \mid \exists \bar{Y}.B$. Let $\bar{Z}$ be the set of free variables in H^k and H^r; all the variables in B are elements of $\bar{Z} \cup \bar{Y}$. The declarative semantics of such rule is defined as

$$\forall \bar{Z}.(H^k \wedge G) \supset (H^r \leftrightarrow \exists \bar{Y}.B)$$

The declarative semantics of a rule $H^k \setminus H^r \iff G \mid \nabla \bar{X}.\exists \bar{Y}.B$ looks similar,

$$\forall \bar{Z}.(H^k \wedge G) \supset (H^r \leftrightarrow \nabla \bar{X}.\exists \bar{Y}.B)$$

However, notice that in a "classical" rule the quantified variables range over terms, whereas in CHR$^{\nabla}$ they range over λ-trees. This means that abstraction and application are also allowed by the syntax, and that α, β_0 and η rules relate equivalent constraints.

Theoretical operational semantics: The other common semantics for CHRs is the so-called *theoretical operational semantics* ω_t (Duck *et al.* 2004), akin to a small-step

https://doi.org/10.1017/S1471068417000230 Published online by Cambridge University Press

Solve. For each built-in constraint c,

$$\langle \{c\} \uplus G, S, B, T, \mathcal{N} \rangle \rightsquigarrow \langle G, S, c \wedge B, T, \mathcal{N} \rangle$$

Introduce. For each constraint c, given a fresh identifier i,

$$\langle \{c\} \uplus G, S, B, T, \mathcal{N} \rangle \rightsquigarrow \langle G, \{c\#i\} \cup S, B, T, \mathcal{N} \rangle$$

Apply. If the set of rules contains a rule named r defined as

$$H^k \setminus H^r \iff G \mid \nabla \bar{X}.\exists \bar{Y}.C$$

and there is a matching L_λ-substitution θ such that $H_1 \equiv \theta(H^k)$, $H_2 \equiv \theta(H^r)$, the set of built-ins B implies $\theta(G)$, that is, $B \Vdash \theta(G)$ and $t = \langle \mathsf{id}(H_1), \mathsf{id}(H_2), r \rangle \notin T$,

$$\langle G, H_1 \uplus H_2 \uplus S, B, T, \mathcal{N} \rangle \rightsquigarrow \langle C' \uplus G, H_1 \cup S, \theta \wedge B, T \cup \{t\}, \mathcal{N} \cup \bar{N} \rangle$$

where C' results from replacing $\bar{X}$ by fresh nominal constants $\bar{N}$, $\bar{Y}$ by fresh variables, and by β_0-reducing as much as possible.

Fig. 2. Theoretical operational semantics of CHR$^\nabla$.

operational semantics. In this case, each rule gives rise to a transition between execution states. Each of these execution states is of the form $\langle G, S, B, T, \mathcal{N} \rangle$, where G is the set of *goal* constraints, S is the constraint *store*, which saves constraints along with an identifier, B is a set of *built-in* constraints, T is the *propagation* history and $\mathcal{N}$ is the set of nominal constants in use. Built-in constraints are those known internally to the CHR engine and for which the engine may perform reasoning; we assume that an entailment relation $\Vdash$ is given for such constraints.

The rules defining the operational semantics for CHR$^\nabla$ are given in Figure 2. They are quite similar to the original ω_t.

- The Solve rule moves a built-in constraint c from the goal set to the built-in set. In practice, this means that the underlying procedure for c is invoked.
- The Introduce step assigns new identifiers to yet-unsolved goals. Attaching such an identifier is necessary to prevent trivial non-termination arising from using the same rule over the same set of constraints repeatedly.
- The Apply rule executes a rule with a matching set of constraints. This is where the differences with "classical" CHRs arise. First, unification is higher order. Second, since we have two types of quantification in the body C, we must introduce new nominal constants and new variables. We need to keep track of which nominal constants have been introduced in order to deal with confluence (Section 3.2), so we introduce a set $\mathcal{N}$ in the execution state to hold that information. Finally, we might need to β_0-reduce some of the obtained constraints in order to put them in the right syntax for further transitions.

Note that we require that constraints C' obtained after freshening and reduction to be in normal form. That is, they *must* be headed by a constraint constructor; abstraction and application are not allowed at constraint level.

3.1 Simple type inference for higher-rank types

Higher-rank types extend Hindley–Damas–Milner types by allowing polymorphic types as arguments to function types. The type $\forall a.(a \to a) \to \mathit{Int}$ is not higher-rank,

https://doi.org/10.1017/S1471068417000230 Published online by Cambridge University Press

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau \rightsquigarrow \top} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \rightsquigarrow C_1 \qquad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow C_2 \qquad \alpha \text{ and } \beta \text{ fresh}}{\Gamma \vdash e_1\ e_2 : \beta \rightsquigarrow \tau_1 \leqslant fn(\alpha, \beta), \tau_2 \leqslant \alpha, C_1, C_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \rightsquigarrow C}{\Gamma \vdash \lambda(x : \tau_1).e : fn(\tau_1, \tau_2) \rightsquigarrow C} \qquad \frac{\Gamma, x : \alpha \vdash e : \tau_2 \rightsquigarrow C \qquad \alpha \text{ fresh}}{\Gamma \vdash \lambda x.e : fn(\alpha, \tau_2) \rightsquigarrow C}$$

Fig. 3. Constraint generation for λ-calculus.

since quantification is at the top of the type. In contrast, $(\forall a.a \to a) \to Int$ is a higher-rank type, since $\forall a.a \to a$, a quantified type, is the type of the argument of the function.

Inference in the presence of such types has led to different approaches (Peyton Jones *et al.* 2007; Dunfield and Krishnaswami 2013). Our aim in this section is to present a simple algorithm that shows the feasibility of using CHR$^\forall$ for encoding such type systems. The focus in the presentation is simplicity, although we conjecture that a simple variation of the presented procedure is complete.

First of all, we need to describe the syntax of the terms and constraints we deal with. In this case, terms will correspond to shapes of types:

- Types headed by constructors are represented as *con* $(C, Args)$, where C is the name of the constructor and *Args* a list of arguments. For example, the type *Maybe Int* is represented as *con* (`"Maybe"`, [*con* (`"Int"`, [])]). For the sake of conciseness in the examples, we use special syntax for function types, *fn* (A, B).
- Polymorphic types are wrapped into a *forall* term constructor, and have as only argument an abstraction binding a variable. For example, $\forall a.a \to a$ is represented as *forall* $(\lambda A.fn\ (A, A))$.
- Type variables are represented by CHR-level variables.

Constraints take types as arguments: They can either be *instantiation* constraints $T_1 \leqslant T_2$, meaning that T_1 is more polymorphic than T_2, or *equality* constraints $T_1 = T_2$. We assume that equality constraints are built in, and perform unification on their arguments.

Constraint-based type inference (Pottier and Rémy 2005; Hage and Heeren 2009) is structured in multiple phases. The first phase is constraint *generation*: Traversing the abstract syntax tree of the expression, we are interested in typing and obtaining the corresponding constraints. We focus on a typed λ-calculus, whose generation judgment $\Gamma \vdash e : \tau \rightsquigarrow C$ is given in Figure 3. This judgment takes as input an environment Γ and an expression e and produces a type τ and some constraints C. The only unusual feature of the presented λ-calculus is the existence of an annotated abstraction. Such a feature is needed to type functions with higher-rank arguments (Peyton Jones *et al.* 2007).

The second phase is constraint *solving*: at this point CHR$^\forall$ enters the game. We implement the solving procedure as a set of rules, given in Figure 4, to be applied exhaustively to the generated constraints. The first rule implements reflexivity. The reader might wonder whether $T \leqslant T$ is a well-defined constraint: It is so, since the duplicated variable appears as argument to a constructor $\leqslant$, not as argument to a free variable. The next two rules simplify instantiations in which the left-hand

https://doi.org/10.1017/S1471068417000230 Published online by Cambridge University Press

$$
\begin{array}{rcl@{\qquad}rcll}
T & \leqslant & T & \Longleftrightarrow & true & & & \\
con(C_1, Args_1) & \leqslant & T_2 & \Longleftrightarrow & con(C_1, Args_1) & = & T_2 & \\
fn(S_1, T_1) & \leqslant & T_2 & \Longleftrightarrow & fn(S_1, T_1) & = & T_2 & \\
forall(Q) & \leqslant & T_2 & \Longleftrightarrow & \exists V.\, Q\ V & \leqslant & T_2 & \text{if } T_2 \not\equiv forall(R) \\
T_1 & \leqslant & forall(Q) & \Longleftrightarrow & \nabla V.T_1 & \leqslant & Q\ V &
\end{array}
$$

Fig. 4. Rules for solving type inference constraints.

side is not polymorphic to an equality. Note that by choosing these rules, we make function types invariant. Finally, polymorphic types are dealt with by generating fresh variables or nominal constants.

After solving, leftover constraints are interpreted. Some of them, like $V = T$, symbolize the type assignment found by the type checker, whereas other types of constraints are expected to be absent in the final set.

Let us check how the type engine proceeds with an expression such as *id* 3, for the common $id :: \forall a.\, a \to a$ function. The generation derivation is

$$
\frac{\Gamma \vdash id : forall(\lambda A.fn(A, A)) \rightsquigarrow \top \qquad \Gamma \vdash 3 : con(\texttt{"Int"}, []) \rightsquigarrow \top}{\Gamma \vdash id\ 3 : T \rightsquigarrow forall(\lambda A.fn(A, A)) \leqslant fn(S, T), con(\texttt{"Int"}, []) \leqslant S}
$$

The first constraint is simplified by instantiating a fresh variable R to $fn\ (R, R) \leqslant fn\ (S, T)$. Then, the instantiation constraint is turned into an equality, effectively unifying all of R, S and T. The second constraint is now $con\ (\texttt{"Int"}, []) \leqslant R$, which is also simplified to an unification $con\ (\texttt{"Int"}, []) = R$. At this point, no instantiation constraint is left, and the value of every type variable has been assigned by unification.

3.2 Confluence and termination properties

CHRs are non-deterministic, as discussed in Section 2.1. In contrast with logic languages such as Prolog, CHRs feature committed choice, that is, once a rule has been applied, there is no built-in backtracking mechanism. Thus, an important question when faced with a set of CHRs is whether they are *confluent*, which means that the same final state is achieved regardless of the order in which rules are applied. The other important question is whether a set of CHRs *terminates* for any given input.

In theory, CHRs could be treated as an instance of an abstract rewriting system. However, the fact that matching involves more than one constraint at once sets them apart from other rewriting systems: Both confluence (Abdennadher 1997; Duck *et al.* 2007) and termination (Pilozzi and De Schreye 2008; Voets *et al.* 2008) require specific techniques. In this section, we study the applicability of those techniques to our setting.

Confluence: In order to define confluence, we first need to define when two execution states are thought of as equivalent. We build on the definition given by Duck *et al.* (2007), which we extend to account for nominal constants.

https://doi.org/10.1017/S1471068417000230 Published online by Cambridge University Press

Definition 2 (*Variants*)
Let $\sigma_1 = \langle G_1, S_1, B_1, T_1, \mathscr{N}_1 \rangle$ and $\sigma_2 = \langle G_2, S_2, B_2, T_2, \mathscr{N}_2 \rangle$ be two execution states. We assume $\mathscr{N}_1$ and $\mathscr{N}_2$ have the same number of variables, otherwise we can just extend the shorter with new fresh ones. For each execution state σ_i, let T_i' be the set of tokens from T_i that mention any of the constraints in S_i and let $\mathscr{V}_i$ be the set of variables appearing in G_i, S_i, B_i or T_i. We say that σ_1 and σ_2 are *variants*, $\sigma_1 \approx \sigma_2$, if either

- there exists a unifier ρ of S_1, S_2, G_1, G_2, T_1' and T_2', and a permutation π between $\mathscr{N}_2$ and $\mathscr{N}_1$ – that is, a bijective mapping between the two sets of nominal constants – such that $\exists \mathscr{V}_1. B_1 \supset \exists \mathscr{V}_1. \rho \wedge \pi(B_2)$ and $\exists \mathscr{V}_2. B_2 \supset \exists \mathscr{V}_2. \rho \wedge \pi^{-1}(B_1)$. In other words, there exists a unifier modulo renaming,
- or both B_1 and B_2 are logically inconsistent.

The idea of being equal modulo permutation comes from the usage of the IDπ rule introduced in Section 2.3. In a logical sense, two execution states that are variants satisfy

$$\nabla \mathscr{N}_1. \exists \mathscr{V}_1. B_1 \supset \nabla \mathscr{N}_1. \exists \mathscr{V}_2. \rho \wedge B_2 \quad \text{and} \quad \nabla \mathscr{N}_2. \exists \mathscr{V}_2. B_2 \supset \nabla \mathscr{N}_2. \exists \mathscr{V}_1. \rho \wedge B_1$$

However, the reading using permutations is more operational.

Once the notion of variance is settled, we can formally define the confluence property. In the following, $\rightsquigarrow$ refers to the theoretical operational semantics relation defined in Figure 2, and $\rightsquigarrow^*$ to its transitive and reflexive closure.

Definition 3 (*Joinable states, local confluence, confluence*)
Given two execution states σ_1 and σ_2, we say that they are *joinable*, $\sigma_1 \downarrow \sigma_2$, if there exists σ_1' and σ_2' such that $\sigma_1 \rightsquigarrow^* \sigma_1'$, $\sigma_2 \rightsquigarrow^* \sigma_2'$ and the end states are variants, $\sigma_1' \approx \sigma_2'$.

A set of CHRs is said to be *locally confluent* if for any states σ_0, σ_1 and σ_2 such that $\sigma_0 \rightsquigarrow \sigma_1$ and $\sigma_0 \rightsquigarrow \sigma_2$, we have that σ_1 and σ_2 are joinable.

A set of CHRs is said to be *confluent* if for any states σ_0, σ_1 and σ_2 such that $\sigma_0 \rightsquigarrow^* \sigma_1$ and $\sigma_0 \rightsquigarrow^* \sigma_2$, we have that σ_1 and σ_2 are joinable.

Note the difference between the two notions of confluence. In local confluence, we take *one* step and then try to join the new states, whereas in normal confluence, we are allowed to take any number of steps in the hypothesis.

Lemma 1 (*Newman 1942*)
If a terminating abstract rewrite system is locally confluent, then it is confluent.

The classical way to prove confluence of CHRs is to determine that your rules are locally confluent and then use Newman's lemma. Proving local confluence directly is still hard, though. Most works (Abdennadher 1997; Duck *et al.* 2007) provide sufficient conditions for a set of CHRs to be locally confluent via the notion of a *critical pair*. Intuitively, a critical pair is a minimal description of a point where confluence is at risk, for example, because more than one rule may apply. As previously, we follow Duck *et al.* (2007).

https://doi.org/10.1017/S1471068417000230 Published online by Cambridge University Press

Definition 4 (*Critical pair*)
Given two rule instances $H_1^k \setminus H_1^r \iff g_1 \mid \nabla\overline{X}_1.\exists\overline{Y}_1.B_1$ and $H_2^k \setminus H_2^r \iff g_2 \mid \nabla\overline{X}_2.\exists\overline{Y}_2.B_2$, we define the following multisets of constraints, using $\uplus$ for disjoint unions:

$$H_1^k \uplus H_1^r = H_1^\cap \uplus H_1^\Delta \quad \text{and} \quad H_2^k \uplus H_2^r = H_2^\cap \uplus H_2^\Delta$$

Intuitively, all the constraints involved in a rule, $H_i^k \uplus H_i^r$, are divided into two sets, $H_i^\cap$ and $H_i^\cap$. The constraints in $H_1^\cap$ and $H_2^\cap$ are those where the rules may overlap – if they do not, the equality in the definition of the critical pair becomes false. H_1^Δ and H_2^Δ, on the other hand, represent those constraints where rules do not coincide. By ranging over all possible partitions of $H_i^\cap$ and H_i^Δ, we consider all possible scenarios in which both rules could apply.

We define the propagation history T_{CP} to include a token $e_i = \langle \mathsf{id}(H_i^k), \mathsf{id}(H_i^r), r_i\rangle$ for each propagation rule. A *critical pair* for these two rules is the pair of execution states:

$$\langle B_1, (H_1^\Delta \uplus H_2^\Delta \uplus H_1^\cap) - H_1^r, H_1^\cap = H_2^\cap \wedge g_1 \wedge g_2, T_{CP} - \{e_1\}, \mathscr{N}_1 \cup \mathscr{N}_2 \cup \overline{X}_1\rangle$$
$$\langle B_2, (H_1^\Delta \uplus H_2^\Delta \uplus H_1^\cap) - H_2^r, H_1^\cap = H_2^\cap \wedge g_1 \wedge g_2, T_{CP} - \{e_2\}, \mathscr{N}_1 \cup \mathscr{N}_2 \cup \overline{X}_2\rangle$$

where $\mathscr{N}_i$ is the set of nominal constants in each rule instance.

A critical pair encodes the result of applying the two rules starting from an initial execution state to which both rules may be applied.

Theorem 1 (*Joinable critical pairs* $\implies$ *locally confluent*)
Given a set $\mathscr{C}$ of CHRs, if all critical pairs are joinable, then $\mathscr{C}$ is locally confluent.

Proof
See the appendix.

Corollary 1 (*Joinable critical pairs + terminating* $\implies$ *confluent*)
Given a terminating set $\mathscr{C}$ of CHRs, if all critical pairs are joinable, then $\mathscr{C}$ is confluent.

As an example of how to apply this technique, let us look at a critical pair for the CHRs in Figure 4. In particular, when instantiation involves a type of the form *forall* $(\lambda X_1.$*forall* $(\lambda X_2.\ \ldots$*forall* $(\lambda X_n.Q)))$, both the first rule, which encodes reflexivity, and the last rule, which tells us what to do when the right-hand side is a quantified type, are applicable. In the first case, the resulting state is $\top$, and in the second case,

$$\textit{forall}\ (\lambda X_1.\textit{forall}\ (\lambda X_2.\ \ldots \textit{forall}\ (\lambda X_n.Q))) \leqslant \textit{forall}\ (\lambda X_2.\ \ldots \textit{forall}\ (\lambda X_n.Q\ \mathtt{a}_1))$$

We need to check that these two states are joinable. The first resulting state does not allow more rules to be applied, but for the second we can keep going until we end up with a constraint of the form $Q\ \alpha_1\ \alpha_2 \ldots \alpha_n = Q\ \mathtt{a}_1\ \mathtt{a}_2 \ldots \mathtt{a}_n$ for fresh existential variables α_i and fresh constants $\mathtt{a}_i$. Reading the definition of variants, we need to prove that

$$\top \supset \exists\overline{\alpha}_i.Q\ \alpha_1\ \ldots\ \alpha_n = Q\ \mathtt{a}_1\ \ldots\ \mathtt{a}_n \quad \text{and} \quad \exists\overline{\alpha}_i.Q\ \alpha_1\ \ldots\ \alpha_n = Q\ \mathtt{a}_1\ \ldots\ \mathtt{a}_n \supset \top$$

https://doi.org/10.1017/S1471068417000230 Published online by Cambridge University Press

The second one is trivial. For the first one, just take α_i equal to a_i and we are done. Since we have not introduced any constant, we can take any permutation of the a_i's to make the execution states variants.

Termination: Most approaches to termination of CHRs are based on a quantity that decreases after each step of solving. A norm is defined by Frühwirth (2000), Voets *et al.* (2008) or Pilozzi and De Schreye (2008) as a function $\|\cdot\| : \mathit{Term} \to \mathbb{N}$ and a level mapping as a function $|\cdot| : \mathit{Constraint} \to \mathbb{N}$. If some conditions over the rules and initial constraints referring to those mappings are satisfied, then the CHRs are terminating for a given initial set of constraints.

A key property in most realizations of these conditions is that of a *rigid* level mapping. A rigid $|\cdot|$ is invariant under substitution: For all constraints C and substitutions θ, $|C| = |\theta C|$. We can redefine this notion in such a way that the techniques of the papers Frühwirth (2000), Voets *et al.* (2008) or Pilozzi and De Schreye (2008) keep working:

- Level mappings assume that constraints are β_0-reduced before application.
- Substitutions θ map variables to possibly higher-order terms.
- The mapping must be independent of nominal constraints. Formally, for every permutation of variables π, $|C| = |\pi(C)|$. Otherwise, a rule like $c(X) \iff \nabla Y\,.\,c(Y)$ may generate an infinite number of new constants.

Using these ingredients, Voets *et al.* (2008) show that the following ranking condition is enough to guarantee termination. Frühwirth (2000) and Pilozzi and De Schreye (2008) describe different conditions in the same spirit.

Definition 5 (*Call set*)
Given a set of CHRs R and an initial set of constraints I, the *call set* $\mathsf{Call}(R, I)$ is the union, over all possible traces of computations of R on I, of all constraints added to the store at every rule application.

Definition 6 (*Ranking condition for CHRs*)
A set of CHRs R and an initial set of constraints I are said to satisfy the *ranking condition* with respect to a level mapping $|\cdot|$ if every constraint in $\mathsf{Call}(R, I)$ is rigid with respect to $|\cdot|$, and for each rule R, matching substitution θ and answer substitution φ:

- If R is a propagation rule $H_1, \ldots, H_n \implies G \mid B_1, \ldots, B_m$, for every $i = 1, \ldots, n$ and $j = 1, \ldots, m$ such that B_j is not built-in, then $|\theta H_i| > |\varphi\theta B_j|$.
- If R is a simpagation rule $H \setminus H_1, \ldots, H_n \implies G \mid B_1, \ldots, B_m$, define

$$p = \max\{|\theta H_1|, \ldots, |\theta H_n|, |\varphi\theta B_1|, \ldots, |\varphi\theta B_m| \mid B_j \text{ not built-in}\},$$

  then the number of constraints with level value p in $\{\theta H_1, \ldots, \theta H_n\}$ is higher than the number of those constraints in $\{\varphi\theta B_1, \ldots, \varphi\theta B_m\}$.

As an example, in the rules for type inference given in Figure 4, the only problematic rule with respect to nominal constants is the last one, since the others do not introduce any constants. We can use a norm which puts equality constraints before instantiation constraints, and for the latter type, adds up the number of *forall*

https://doi.org/10.1017/S1471068417000230 Published online by Cambridge University Press

constituents from both sides. The reader can check that this norm always decreases during solving.

4 Implementation

We have built a solver for CHR^{∇} as a deeply embedded language in Haskell; it is available at `https://git.science.uu.nl/f100183/uchrp`. Apart from the features described in this paper, it also provides a type-safe interface to CHRs and support for rule priorities as described by Koninck *et al.* (2007). The only difference between the library and this paper is that the former uses a built-in predicate to introduce variables, instead of making the operator ∇ part of the syntax of rules.

Using this library, we have implemented the type checker featured in this paper to show the feasibility of the approach; it is available at `https://git.science.uu.nl/f100183/quique`. In fact, this type checker also implements impredicative polymorphism, although the description of the corresponding set of CHRs is out of scope for this paper. This suggests that CHR^{∇} is expressive enough for encoding complex type systems.

Our preliminary evaluation shows that CHR^{∇} is competitive in terms of performance with similar approaches. Furthermore, there is room for optimizations such as performing on-the-fly substitution and indexing constraints in the solver.

5 Related work

λ-tree syntax and ∇ quantification: The closest system to ours is λProlog (Miller 1991; Miller and Nadathur 2012). We have drawn inspiration from several of its features, like the use of λ-tree syntax and pattern unification in heads. The main difference is that λProlog, as its name suggests, builds upon the logic programming paradigm, so it includes search with backtracking. On the other hand, CHR^{∇} embodies committed choice: No backtracking is done once a rule is chosen to apply. As a result, the kind of formulas we can represent in CHR^{∇} is more limited than in λProlog: The latter allows hereditary Harrop formulas, where both universal quantification and implication may appear in the left-hand side of an implication, whereas we do not allow left-nested implications.

The Abella theorem prover (Baelde *et al.* 2014) was developed to reason about λProlog and inherits many of its features. Abella uses the ∇ quantifier to reason about the object logic into its meta-logic. Our approach is similar, but we have blurred the lines between object and meta-logic: In CHR^{∇}, only universal quantification in using nominal constants is available. This results in a more uniform approach to deal with constraints.

In the functional world, ML has been extended to support binders using an approach similar to λ-tree syntax (Miller 1990). A type $a \Rightarrow b$ represents values of type b where a variable of type a is bound. Matching on a value of this type binds to a function.

https://doi.org/10.1017/S1471068417000230 Published online by Cambridge University Press

Nominal abstract syntax: Another different approach to introduce binders in the syntax of a language is given by *nominal syntax*. The main difference is that correctness is based on the notion of *variable swapping*. Using this idea, it can be proven that programs operate correctly under α-equivalence. See Tiu (2006) for a detailed account of the differences.

αProlog extends Horn clauses with nominal binding (Cheney and Urban 2008). FreshML (Shinwell *et al.* 2003) implements these ideas in the functional world.

6 Conclusion and future work

CHR^{∇} provides an extension to CHRs in which binding manipulation is reflected in the syntax. This provides a sound basis for developing compilers and type checkers for advanced type programming languages.

In the future, we aim to look at other extensions of CHRs and their integration with binding and ∇ quantification. In particular, $CHR^{\vee}$ (Abdennadher and Schütz 1998), which features disjunction, enables us to escape from the committed choice semantics of CHR and explore several branches of computation. Such an ability is needed for type checking some forms of overloading, like the one present in Swift (Swift Team 2016).

References

Abdennadher, S. 1997. Operational semantics and confluence of constraint propagation rules. In *Proc. of CP97*, Linz, Austria, October 29–November 1, G. Smolka, Ed.

Abdennadher, S. and Schütz, H. 1998. $CHR^{\vee}$: A flexible query language. In *Proc. of International Conference on Flexible Query Answering Systems, FQAS '98*, 1–14.

Baelde, D., Chaudhuri, K., Gacek, A., Miller, D., Nadathur, G., Tiu, A. and Wang, Y. 2014. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning 7* (2), 1–89.

Charguéraud, A. 2012. The locally nameless representation. *Journal of Automated Reasoning 49,* 3, 363–408.

Cheney, J. and Urban, C. 2008. Nominal logic programming. *ACM Transactions on Programming Languages and Systems 30* (5), 26:1–26:47.

Csorba, J., Zombori, Z. and Szeredi, P. 2012. Pros and Cons of Using CHR for Type Inference. Unpublished, available in the first author's web page.

de Bruijn, N. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings) 75,* 5, 381–392.

Dijkstra, A., van den Geest, G., Heeren, B. and Swierstra, S. D. 2007. *Modelling Scoped Instances with Constraint Handling Rules.* Technical Report, Department of Information and Computing Sciences, Utrecht University.

Duck, G. J., Stuckey, P. J., de la Banda, M. G. and Holzbaur, C. 2004. The refined operational semantics of constraint handling rules. In *Proc. of ICLP 2004*, Saint-Malo, France, September 6–10, 90–104.

Duck, G. J., Stuckey, P. J. and Sulzmann, M. 2007. Observable confluence for constraint handling rules. In *Proc. of ICLP 2007*, Porto, Portugal, September 8–13, 2007, V. Dahl and I. Niemelä, Eds., 224–239.

https://doi.org/10.1017/S1471068417000230 Published online by Cambridge University Press

Dunfield, J. and Krishnaswami, N. R. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proc. of International Conference on Functional Programming, ICFP'13*, G. Morrisett and T. Uustalu, Eds. ACM, 429–442.

Eisenberg, R. A., Weirich, S. and Ahmed, H. G. 2016. Visible type application. In *Programming Languages and Systems 25th European Symposiun on Programming – ESOP 2016*, Lecture Notes in Computer Science 9632. Springer, 229–254.

Frühwirth, T. 1998. Theory and practice of constraint handling rules. *Journal of Logic Programming 37,* 13, 95–138.

Frühwirth, T. 2000. *Proving Termination of Constraint Solver Programs*. Springer, Berlin, Heidelberg, 298–317.

Frühwirth, T. 2009. *Constraint Handling Rules*, 1st ed. Cambridge University Press.

Hage, J. and Heeren, B. 2009. Strategies for solving constraints in type and effect systems. *Electronic Notes in Theoretical Computer Science 236*, 163–183.

Huet, G. 1975. A unification algorithm for typed λ-calculus. *Theoretical Computer Science 1,* 1, 27–57.

Koninck, L., Schrijvers, T. and Demoen, B. 2007. User-definable rule priorities for CHR. In *Proc. of International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP '07*. ACM, 25–36.

Miller, D. 1990. *An Extension to ML to Handle Bound Variables in Data Structures*. Technical Report, Department of Computer and Information Science, University of Pennsylvania.

Miller, D. 1991. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation 1*, 253–281.

Miller, D. 2000. In *Abstract Syntax for Variable Binders: An Overview*, *Proceeding of Computational Logic 2000*, Lecture Notes in Computer Science 1861. Springer, 239–253.

Miller, D. and Nadathur, G. 2012. *Programming with Higher-Order Logic*, 1st ed. Cambridge University Press, New York.

Miller, D. and Tiu, A. 2005. A proof theory for generic judgments. *ACM Transactions on Computational Logic 6* (4), 749–783.

Peyton Jones, S., Vytiniotis, D., Weirich, S. and Shields, M. 2007. Practical type inference for arbitrary-rank types. *Journal of Functional Programming 17,* 1, 1–82.

Pfenning, F. and Elliott, C. 1988. Higher-order abstract syntax. In *Proc. of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation, PLDI '88*. ACM, New York, 199–208.

Pilozzi, P. and De Schreye, D. 2008. Termination analysis of CHR revisited. In *Proc. of ICLP 2008*, Udine, Italy, December 9–13, M. Garcia de la Banda and E. Pontelli, Eds. 501–515.

Pottier, F. and Rémy, D. 2005. The essence of ML type inference. In *Advanced Topics in Types and Programming Languages*, B. C. Pierce, Ed. MIT, Chapter 10, 389–489.

Qian, Z. 1993. *Linear Unification of Higher-Order Patterns*, 391–405.

Serrano, A. and Hage, J. 2016. Type error diagnosis for embedded DSLs by two-stage specialized type rules. In *Programming Languages and Systems 25th European Symposium on Programming – ESOP 2016*, Lecture Notes in Computer Science 9632. Springer, 672–698.

Shinwell, M. R., Pitts, A. M. and Gabbay, M. J. 2003. FreshML: Programming with binders made simple. In *Proc. of ACM SIGPLAN International Conference on Functional Programming, ICFP '03*. ACM, 263–274.

Sneyers, J., Schrijvers, T. and Demoen, B. 2009. The computational power and complexity of constraint handling rules. *ACM Transactions on Programming Languages and Systems 31* (2), 8:1–8:42.

https://doi.org/10.1017/S1471068417000230 Published online by Cambridge University Press

Sneyers, J., Weert, P. V., Schrijvers, T. and Koninck, L. 2010. As time goes by: Constraint Handling Rules. *Theory and Practice of Logic Programming 10,* 1, 1–47.

Stuckey, P. J., Sulzmann, M. and Wazny, J. 2006. Type processing by constraint reasoning. In *Programming Languages and Systems*, N. Kobayashi, Ed. Lecture Notes in Computer Science, Vol. 4279. Springer, 1–25.

Sulzmann, M., Duck, G. J., Peyton Jones, S. and Stuckey, P. J. 2007. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming 17,* 1, 83–129.

Sulzmann, M., Wazny, J. and Stuckey, P. J. 2006. A framework for extended algebraic data types. In *Proc. of International Symposium on Functional and Logic Programming, FLOPS '06*, 47–64.

Swift Team. 2016. Type checker design and implementation. Available at `https://github.com/apple/swift/blob/master/docs/TypeChecker.rst`.

Tiu, A. 2006. A logic for reasoning about generic judgments. In *Proc. of 1st International Workshop on Logical Frameworks and Meta-Languages, LFMTP '06. Electronic Notes in Theoretical Computer Science 174* (5), 3–18.

Voets, D., Pilozzi, P. and De Schreye, D. 2008. A new approach to termination analysis of constraint handling rules. In Pre-proceedings of LOPSTR 2008, pp. 28–42.

Vytiniotis, D., Peyton Jones, S., Schrijvers, T. and Sulzmann, M. 2011. OutsideIn(X): Modular type inference with local assumptions. *Journal of Functional Programming 21* (4–5), 333–412.

Wazny, J. 2006. *Type Inference and Type Error Diagnosis for Hindley/Milner with Extensions.* PhD Thesis, University of Melbourne, Australia.

Appendix A. Proof of the confluence theorem

Theorem 1 (Joinable critical pairs $\Longrightarrow$ locally confluent) Given a set $\mathscr{C}$ of CHRs, if all critical pairs are joinable, then $\mathscr{C}$ is locally confluent.

Proof

The proof is similar to the one in the paper by Abdennadher (1997); we need to consider each possible pair of execution steps in the theoretical operational semantics ω_t.

Solve + Solve: In this case, we start with an execution state $\langle \{c_1, c_2\} \uplus G, S, B, T, \mathscr{N} \rangle$ for which two different built-ins are moved into the third component:

$$\langle \{c_2\} \uplus G, S, \{c_1\} \wedge B, T, \mathscr{N} \rangle \quad \text{and} \quad \langle \{c_1\} \uplus G, S, c_2 \wedge B, T, \mathscr{N} \rangle$$

Now, we can choose to move the other built-in to restore variance (actually, equality) of the execution states.

Introduce + Introduce, Solve + Introduce: It is similar to the Solve + Solve case, since the variance of the execution states is independent of the other in which we perform those steps. As in the previous case, we have equality of the joined states.

Solve + Apply, Introduce + Apply: By inspection of the rules for the semantics, we see that the constraints affected by Solve and Introduce come from the G component of the execution state, whereas those affected by Apply come from S. This means that it is not possible for the execution steps to affect the same constraint. As a result,

https://doi.org/10.1017/S1471068417000230 Published online by Cambridge University Press

if we take a step in one direction, we can always apply the other one irrespectively of the former, leading to variant states.

Apply + Apply: This is the interesting case. Suppose that the following rule instances have been applied:

$$H_1^k \setminus H_1^r \iff g_1 \mid \nabla \overline{X}_1.\, \exists \overline{Y}_1.\, B_1 \quad \text{and} \quad H_2^k \setminus H_2^r \iff g_2 \mid \nabla \overline{X}_2.\, \exists \overline{Y}_2.\, B_2$$

Following Abdennadher (1997), we further distinguish between two scenarios.

- No constraint in the first rule unified with another in the second rule (disjoint peaks). In this case, the application of one rule does not interfere with the application of the other. Thus, given the states S_1 and S_2 resulting from applying the first and second rules, respectively, we can join them by applying the second rule to S_1 and the first rule to S_2.
  The only problem is finding the permutation between the nominal constants introduced by each rule. Take $\mathscr{N}_1$ to be the set of constants introduced by the first rule in the first state, $\mathscr{N}_2$ those introduced then by applying the second rule, and the same for $\mathscr{M}_1$ and $\mathscr{M}_2$ with respect to S_2. We just need to permute $\mathscr{N}_1$ to $\mathscr{M}_1$ and $\mathscr{N}_2$ to $\mathscr{M}_2$ to satisfy the variance requirements.
- At least one constraint in the first rule unifies with another in the second rule (critical peaks). In this case, the condition on joinability of critical pairs is enough to guarantee joinability of the corresponding execution states: all the other constraints not mentioned in the critical pair remain untouched by the execution of the rules.

□

https://doi.org/10.1017/S1471068417000230 Published online by Cambridge University Press