Canonical Regular Types

Ethan K. Jackson[1], Nikolaj Bjørner[1], and Wolfram Schulte[1]

1 **Microsoft Research, Redmond, WA**
`ejackson, nbjorner, schulte@microsoft.com`

Abstract

Regular types represent sets of structured data, and have been used in logic programming (LP) for verification. However, first-class regular type systems are uncommon in LP languages. In this paper we present a new approach to regular types, based on *type canonization*, aimed at providing a practical first-class regular type system.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases Regular types, Canonical forms, Type canonizer

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Regular types describe (infinite) sets of structured data, and have a long history in the verification of logic programs. Briefly, a *data term* is a term t built up from a universe of constants using *data constructors*:

$$t \doteq employee(name(\text{``}John\text{''}, \text{``}Smith\text{''}), id(100))$$

The functions *name*, *employee*, and *id* are constructors for building data; the remaining symbols are constants. A *type term* is a term τ denoting a set of data terms:

$$\tau \doteq employee(name(\text{``}John\text{''}, \mathsf{String}), id(\mathsf{Natural}))$$

The special constants String and Natural denote the sets of all strings and natural numbers. In this example, τ denotes the set of all employee data terms where the employee's first name is "*John*" and the employee's ID is a natural number. We write $[\![\tau]\!]$ for the set of data terms denoted by τ:

$$[\![\tau]\!] \doteq \{employee(name(\text{``}John\text{''}, x), id(y)) \mid x \in [\![\mathsf{String}]\!] \wedge y \in \mathbb{Z}_+ \cup \{0\}\}$$

In this paper we develop a language of type terms to represent, manipulate and canonize regular types. We use the phrase *type term* and *regular type* interchangeably.

Regular types have been primarily used to verify properties of untyped logic programs using the following technique [7]: For each n-ary program relation r define an n-ary data constructor f_r. Compute a type term τ_r such that if $r(x_1, \ldots, x_n)$ holds in the program, then the data term $f_r(x_1, \ldots, x_n)$ is a member of $[\![\tau_r]\!]$. Now, τ_r can be used to check properties of r using type-theoretic operations of *type equality* ($\approx$) and *subtype testing* ($<:$). For example, if $[\![\tau_r]\!]$ is empty then r never holds in the program, which is likely a mistake.

However, unlike other typing paradigms, regular types have not been embraced as a first-class type system in logic programming. Their application remains narrowly scoped to verification of untyped logic programs. There are several reasons why regular types are difficult to use at the language level:

No Language-Level Representation Regular types are equivalent to powerful recognizers on data terms, called *non-deterministic tree automata* (NDTAs). Most verification tools use NDTAs to represent types, but this low-level representation is unsuitable for programmer manipulation. Meanwhile, the shallow embedding of regular types via *type programs* [7] does not provide a first-class type system.

No Unique Representation Regardless of representation, regular types are non-unique. Two types may have the same meaning, but look drastically different. Programmers cannot be expected to know if this is the case, because type equality is EXPTIME-complete. This limits the effectiveness of type inference to report useful information.

No Infinite Base Types Existing approaches do not support an infinite universe of constants. This restriction is fundamental to all approaches based on finite tree automata, where it provides key closure properties. Thus, languages with infinite base types, such as the mathematical set of integers, are not supported.

In this paper we present a new approach to regular types aimed at typed logic programs over a first-class regular type system. This approach has been implemented in our LP language FORMULA [1][11]. Our contributions are:

First-class Types and Declarations We define a language for regular types using type terms. Programmers explicitly declare the types of data constructor arguments using type terms. We develop a special class of type declarations, which we call *uniform*, where type equality and subtype testing are in coNP.

Unique Representations Under the uniformity restriction, we develop a *type canonizer* that converts any two semantically equivalent type terms into syntactically identical type terms. In this way, programmers observe the results of type inference as uniquely represented type terms, regardless of the steps taken by type inference. We experimentally show that inferred type terms are small.

New Algorithms/Infinite Base Types We present alternative algorithms based on algebraic manipulation of type terms instead of tree automata. This formalization eliminates the finite signature restriction and supports infinite base types directly.

This paper is organized as follows. Section 2 contains related work. Section 3 formalizes regular types over infinite base types. Section 4 introduces uniform declarations. Section 5 describes the canonization algorithm. We conclude in Section 6.

2 Related Work

Regular types have been primarily been used for verification of untyped logic programs [8, 13]. *Ciao-Prolog* uses a shallow embedding into logic programming [9]. *NU-Prolog* was reported to have first-class regular types [5], while *Mercury* employs a Hindley-Milner style type system [12]. However, other programming paradigms provide support for first-class regular types [10, 3]. Though none of them employ canonical forms to present the results of type inference. Instead, regular types are commonly stored as optimized NDTAs [1]. Further optimizations can be obtained by restricting the class of type declarations [4], which is conceptually similar to our approach. Extensions to regular types include *feature algebras* [2] and may add arrow types [6].

[1] See `http://research.microsoft.com/formula`

3 Regular Types and their Semantics

In this section we develop a theory of regular types through a language of type terms. We start from an algebraic signature Σ_Δ, called the *data signature*, identifying the name/arity of data constructors and the universe U of constants. A data signature contains a finite number of non-nullary *function symbols* and may contain an infinite universe of constants U. Here is an example of a data signature: $(\mathbb{Z} \cup \mathbb{S}, employee(,), name(,), age())$, where $\mathbb{S}$ is the set of all strings. Typically, the language predefines many constants in the universe and the programmer defines the non-nullary function symbols.

A *data term* is either a constant, or an application of a Σ_Δ-function to data terms. Importantly, data terms are *untyped*, meaning any arity-respecting sequence of applications is a legal data term, even if it appears to be nonsensical: $name(id(100), employee(40, \text{“}Foo\text{”}))$ Semantically, data terms are interpreted in a very simple way: Two data terms denote the same object if and only if they are identical sequences of function applications and constants. This interpretation yields the *Herbrand Universe* of Σ_Δ, written $\mathcal{H}(\Sigma_\Delta)$, which is the set of all objects labeled by data terms under this rule of equality. From henceforth, the phrases *data constructor* and Σ_Δ*-function* are equivalent.

Regular types allow the programmer to identify the meaningful data terms by describing subsets of data terms. For example, the regular type $name(\mathsf{String}, \mathsf{String})$ identifies all the data terms with string arguments; the unwanted term $name(id(100), id(100))$ does not belong to this set. In our setting a regular type is type term that can be formed using: (1) data constructors, (2) constants, (3) *base types* such as Integer or String, (4) *type variables* such as α_{cons} or α_{list}, and (5) the operations $\cap$ and $\cup$.

3.1 Type Signatures and Terms

Formally, a type term is a term formed over a *type signature* Σ_τ:

▶ **Definition 1** (Type Signature). A *type signature* Σ_τ is extends a data signature Σ_Δ:

$$\Sigma_\tau \doteq (\Sigma_\Delta, V, B, \bot, \cap, \cup). \qquad (1)$$

1. V is a (possibly infinite) set of nullary functions called the set of type variables. The symbols in V are disjoint from other symbols.
2. B is a finite set of constants called the set of base types. The symbols in B are disjoint from other symbols, except for the distinguished base type $\bot \in B$, called void.
3. The type intersection ($\cap$) operation and type union ($\cup$) operation are binary operations. These operations are not in Σ_Δ, V, or B.

Conventions A type term τ is a term over Σ_τ, and $\mathcal{H}(\Sigma_\tau)$ is the Herbrand Universe of type terms. Let $\sigma, \sigma_1, \sigma_2, \ldots$ range over the nullary symbols of Σ_Δ and $f, g, \ldots$ range over the non-nullary symbols of Σ_Δ. Also, $\alpha, \alpha_1, \ldots$ range over type variables and $\beta, \beta_1, \ldots$ range over base types. We write $f(\overline{\tau})$ as shorthand for $f(\tau_1, \ldots, \tau_n)$. For the remainder of this paper we assume all data constructors are binary functions. However, all definitions and theorems generalize for functions of arbitrary arity.

3.2 Type Environments and Denotations

The meaning of a type term τ is a set of data terms, which is fixed except for the meaning of type variables. For example, $[\![id(\alpha)]\!]$ depends on the denotation of the type variable α. A *type environment* η provides the missing information in the form of a function from type variables to sets of data terms.

▶ **Definition 2** (Type Environments)**.** A *type environment* $\eta : V \to 2^{\mathcal{H}(\Sigma_\Delta)}$ is a function from type variables to sets of data terms. Set-theoretical operations on $2^{\mathcal{H}(\Sigma_\Delta)}$ can be lifted to type environments by defining:

$$\begin{array}{llll} \eta \sqcap \eta' & \doteq \lambda\alpha \ . \ \eta(\alpha) \cap \eta'(\alpha) & \bot_{env} & \doteq \lambda\alpha \ . \ \emptyset \\ \eta \sqcup \eta' & \doteq \lambda\alpha \ . \ \eta(\alpha) \cup \eta'(\alpha) & \top_{env} & \doteq \lambda\alpha \ . \ \mathcal{H}(\Sigma_\Delta). \end{array}$$

▶ **Definition 3** (Type Denotation)**.** A type denotation function $\llbracket \ \rrbracket_\eta$ is a function from type terms and environments to sets of data terms. It gives a denotation to every type term with respect to a fixed environment.

$$\llbracket \ \rrbracket : \mathcal{H}(\Sigma_\tau) \to \left(V \to 2^{\mathcal{H}(\Sigma_\Delta)}\right) \to 2^{\mathcal{H}(\Sigma_\Delta)}. \qquad (2)$$

satisfying:

$$\begin{array}{llll|llll} \llbracket \bot \rrbracket_\eta & \doteq & \emptyset. & & \llbracket \beta \rrbracket_\eta & \doteq & \{\sigma_1, \sigma_2, \ldots\}, & \beta \in B - \{\bot\}. \\ \llbracket \alpha \rrbracket_\eta & \doteq & \eta(\alpha), & \alpha \in V. & \llbracket \tau_1 \cup \tau_2 \rrbracket_\eta & \doteq & \llbracket \tau_1 \rrbracket_\eta \cup \llbracket \tau_2 \rrbracket_\eta. & \\ \llbracket \sigma \rrbracket_\eta & \doteq & \{\sigma\}, & \sigma \in \Sigma_\Delta. & \llbracket \tau_1 \cap \tau_2 \rrbracket_\eta & \doteq & \llbracket \tau_1 \rrbracket_\eta \cap \llbracket \tau_2 \rrbracket_\eta. & \\ & & & & \llbracket f(\tau_1, \tau_2) \rrbracket_\eta & \doteq & \left\{ f(t_1, t_2) \;\middle|\; \begin{array}{l} t_1 \in \llbracket \tau_1 \rrbracket_\eta \wedge \\ t_2 \in \llbracket \tau_2 \rrbracket_\eta \end{array} \right\}, & f \in \Sigma_\Delta. \end{array}$$

The denotations of base types are fixed by the language. In other words, they are independent of the environment, denote unique sets, and are closed under intersection. For all $\beta, \beta', \eta, \eta'$:

$$\llbracket \beta \rrbracket_\eta = \llbracket \beta' \rrbracket_{\eta'} \Leftrightarrow \beta = \beta', \quad \exists \beta'' \ \ \llbracket \beta \rrbracket_\eta \cap \llbracket \beta' \rrbracket_\eta = \llbracket \beta'' \rrbracket_\eta.$$

3.3 Type Variables and Declarations

Type variables have a very different use in regular types compared to other type systems. They are used to define recursive data types via a system of *type equations*. Solutions to these equations are type environments where the equations hold. A *type equation* is a pair of type terms, written $\tau \approx \tau'$. A type equation holds for a type environment η if $\llbracket \tau \rrbracket_\eta = \llbracket \tau' \rrbracket_\eta$. Programmers introduce type variables through *type declarations*, which are equations of the form $\alpha \approx \tau$. The smallest solution to these equations gives a unique type environment fixing the denotation of all variables.

▶ **Definition 4** (Type Declarations)**.** A *set of type declarations* $\mathcal{D}$ is a finite set of type equations of the form $\alpha \approx \tau$, where $\alpha \in V$. For each $\alpha \in V$ there is at most one equation in $\mathcal{D}$ with α on the left hand side.

▶ **Example 5** (Declaration of Integer Lists)**.** The following declarations characterize finite lists of integers:

$$\mathcal{D} \doteq \{\alpha_{cons} \approx cons(\textsf{Integer}, \alpha_{list}),\ \alpha_{list} \approx nil \cup \alpha_{cons}\}, \quad \Sigma_\Delta \doteq (\mathbb{Z}, nil, cons(,)).$$

The solution to this system of equations produces the expected result, because any solution η must have $nil \in \llbracket \alpha_{list} \rrbracket_\eta$, implying $\llbracket cons(\textsf{Integer}, nil) \rrbracket_\eta \subseteq \alpha_{cons}$, implying $\llbracket nil \cup cons(\textsf{Integer}, nil) \rrbracket_\eta \subseteq \alpha_{list}$. By induction α_{list} and α_{cons} obtain their usual denotations. We now formalize this result.

▶ **Definition 6** (Least Environment)**.** A set of type declarations distinguishes a unique type environment $\eta(\mathcal{D})$ and denotation $[\![\ ]\!]_{\eta(\mathcal{D})}$, which is the least environment satisfying all type declarations:

$$\eta(\mathcal{D}) \doteq min\left\{ \eta \in Env(\Sigma_\tau) \;\middle|\; \forall \alpha \approx \tau \in \mathcal{D}, [\![\alpha]\!]_\eta = [\![\tau]\!]_\eta \right\}. \tag{3}$$

▶ **Lemma 7.** *The environment $\eta(\mathcal{D})$ exists and is unique. It is the least fixpoint of a monotone operator $\Gamma : Env(\Sigma_\tau) \to Env(\Sigma_\tau)$.*

▶ **Definition 8** (Models Relation)**.** A set of declarations $\mathcal{D}$ may imply additional type equations. Define:

$$\mathcal{D} \models \tau \approx \tau' \quad \doteq \quad [\![\tau]\!]_{\eta(\mathcal{D})} = [\![\tau']\!]_{\eta(\mathcal{D})}. \tag{4}$$

Conventions The $\models$ relation is read: *"$\mathcal{D}$ models the equation $\tau \approx \tau'$"*. Two sets of declarations are equivalent, written $\mathcal{D} \approx \mathcal{D}'$, if they model the same equations:

$$\mathcal{D} \approx \mathcal{D}' \doteq (\forall \tau \approx \tau' \ \mathcal{D} \models \tau \approx \tau' \Leftrightarrow \mathcal{D}' \models \tau \approx \tau'). \tag{5}$$

For the remainder of this paper we drop the subscript $\eta(\mathcal{D})$ from $[\![\ ]\!]$ when the corresponding $\mathcal{D}$ is clear from context. Similarly, we write $\tau \approx \tau'$ instead of $\mathcal{D} \models \tau \approx \tau'$. The type term τ' is a subtype of τ, written $\tau' <: \tau$, if $[\![\tau']\!] \subseteq [\![\tau]\!]$. Equivalently, $\tau' <: \tau$ if and only if $\mathcal{D}$ models the equation $\tau \cap \tau' \approx \tau'$.

▶ **Lemma 9.** *The type intersection ($\cap$) and union ($\cup$) operations inherit the properties of set-theoretical intersection and union: they are idempotent, commutative, associative, and satisfy distributivity and absorbtion properties. Furthermore, the following identities hold for every $\mathcal{D}$:*

$$\begin{array}{ll}
(\textit{product-}\cap) & f(\tau_1, \tau_2) \cap f(\tau_1', \tau_2') \approx f(\tau_1 \cap \tau_1', \tau_2 \cap \tau_2') \\
(\textit{disjoint-}f, g) & g(\tau_1, \tau_2) \cap f(\tau_1', \tau_2') \approx \bot \quad \textit{when } f \textit{ and } g \textit{ are different.} \\
(\textit{disjoint-}\sigma, f) & \sigma \cap f(\tau_1, \tau_2) \approx \bot \\
(\textit{disjoint-}\beta, f) & \beta \cap f(\tau_1, \tau_2) \approx \bot \\
(\textit{disjoint-}\sigma, \sigma') & \sigma \cap \sigma' \approx \bot \quad \textit{when } \sigma \neq \sigma' \\
(\textit{member}) & \sigma \cap \beta \approx \sigma \quad \textit{when } \sigma \in [\![\beta]\!] \\
(\textit{non-member}) & \sigma \cap \beta \approx \bot \quad \textit{when } \sigma \notin [\![\beta]\!] \\
(\textit{base-}\cap) & \beta \cap \beta' \approx \beta'' \quad \textit{when } [\![\beta]\!] \cap [\![\beta']\!] = [\![\beta'']\!]
\end{array}$$

4 Uniform Declarations

For the remainder of this paper we study type environments generated by a restricted class of type declarations, which we call *uniform declarations*. In order to avoid confusion, let us emphasize that uniformity is a restriction *only* on type declarations $\mathcal{D}$. Arbitrary type terms can be constructed w.r.t. to uniform declarations; as before the denotations of terms continues to be given by $[\![\ ]\!]_{\eta(\mathcal{D})}$. To illustrate this restriction, we begin with an example of declarations that are not uniform:

▶ **Example 10** (Lists of Various Lengths).

$$\begin{aligned}
\alpha_{L2} &\approx \mathit{cons}(\mathsf{Integer}, \mathit{cons}(\mathsf{Integer}, \mathit{nil} \cup \alpha_{L2})).\\
\alpha_{L3} &\approx \mathit{cons}(\mathsf{Integer}, \mathit{cons}(\mathsf{Integer}, \mathit{cons}(\mathsf{Integer}, \mathit{nil} \cup \alpha_{L3}))).\\
\alpha_{L} &\approx \alpha_{L2} \cup \alpha_{L3}.
\end{aligned}$$

The type variables α_{L2} and α_{L3} denote lists with lengths divisible by two and three; α_L denotes their union. These types are related in non-trivial ways: $\alpha_{L2} \cap \alpha_{L3} \not\approx \bot$, $\alpha_{L2} <: \alpha_L$, and $\alpha_{L3} <: \alpha_L$. In general, proving these relationships may require witnesses of exponential size. Exponentially large witnesses can be eliminated (and the complexity class reduced) by restricting the structure of type declarations:

▶ **Definition 11** (Uniform Declarations). A set of declarations $\mathcal{D}$ is uniform if:

1. For every $\alpha \approx \tau \in \mathcal{D}$, either τ is free of constructor applications (application-free), or $\tau = f(\tau_1, \tau_2)$ and every τ_i is application-free.
2. For every $f \in \Sigma_\Delta$ there is exactly on equation of the form $\alpha_f \approx f(\tau_1, \tau_2)$.

Example 5 is uniform, while Example 10 is not. Importantly, the uniformity restriction does not prevent arbitrarily precise approximations of general regular types. The types denoted by α_{L2} and α_{L3} can be arbitrarily approximated by the following type terms over the uniform declaration of integer lists:

▶ **Example 12** (Uniform Approximations of Non-uniform Types).

$$\begin{aligned}
\tau_{L2(1)} &\doteq \mathit{cons}(\mathsf{Integer}, \mathit{cons}(\mathsf{Integer}, \mathit{nil})).\\
\tau_{L3(1)} &\doteq \mathit{cons}(\mathsf{Integer}, \mathit{cons}(\mathsf{Integer}, \mathit{cons}(\mathsf{Integer}, \mathit{nil}))) \ldots\\
\tau_{L2(i)} &\doteq \tau_{L2(i-1)} \cup \mathit{cons}(\mathsf{Integer}, \mathit{cons}(\mathsf{Integer}, \tau_{L2(i-1)})).\\
\tau_{L3(i)} &\doteq \tau_{L3(i-1)} \cup \mathit{cons}(\mathsf{Integer}, \mathit{cons}(\mathsf{Integer}, \mathit{cons}(\mathsf{Integer}, \tau_{L3(i-1)}))) \ldots
\end{aligned}$$

Observe, $\tau_{L2(i)} <: \alpha_{L2} <: \alpha_{list}$, $\tau_{L3(i)} <: \alpha_{L3} <: \alpha_{list}$, and $\tau_{L2(i)} \cap \tau_{L3(i)} \not\approx \bot$.

This example also illustrates that type terms can have arbitrary nesting of function symbols; uniformity only constrains type declarations. Uniform declarations provide two key results that aid in type canonization. First, type canonization is simpler because all variables not of the form α_f can be eliminated from type expressions over uniform declarations. Second, type equations are easier to decide; they become coNP-complete as opposed to EXPTIME-complete. We now elaborate on these results. Note that similar observations have been made for restricted classes of XML schema [4].

4.1 Orientability and Complexity of Uniform Declarations

In this section we show that uniform declarations can be rewritten to eliminate dependencies on type variables that are not of the form $\alpha_f \approx f(\overline{\tau})$. As a result, we say a variable α is an *auxiliary variable* if its declaration is $\alpha \approx \tau$ and τ is application-free. Orienting the declarations allows many simplifying assumptions when canonizing type expressions. Two lemmas are required to prove the orientability of uniform declarations.

▶ **Lemma 13** (Substitution). *If $\{\alpha \approx \tau, \alpha' \approx \tau'\} \subseteq \mathcal{D}$, then any occurrence of α in τ' can be replaced with τ. In symbols:*

$$\mathcal{D} \approx \mathcal{D} \setminus \{\alpha_1 \approx \tau_1\} \cup \{\alpha_1 \approx \tau_1[\alpha_2/\tau_2]\}, \tag{6}$$

where $\tau'[\alpha/\tau]$ is the replacement of every occurrence of α with τ.

▶ **Lemma 14** (Eliminating Self-Dependencies). *If $\alpha \approx \tau \in \mathcal{D}$, τ is application-free, and α appears in τ, then α can be eliminated from τ.*

▶ **Theorem 15** (Uniform Declarations are Orientable). *A set of uniform declarations $\mathcal{D}$ is equivalent to a set of uniform declarations $\widehat{\mathcal{D}}$ where:*

$$\forall \alpha \approx \tau \in \widehat{\mathcal{D}},\ vars(\tau) \subseteq \{\alpha_f \mid f \in \Sigma_\Delta\}. \tag{7}$$

The problem of deciding a type equation $\tau' \approx \tau$ is to decide if $\mathcal{D} \models \tau' \approx \tau$. For instance, we earlier showed that subtype testing $\tau' <: \tau$ is equivalent to deciding a type equation $\tau' \cap \tau \approx \tau'$. Our solution is by canonization; a *type canonizer* converts two type terms to syntactically identical terms if and only if the terms have the same denotation. First, we prove coNP-completeness of deciding type equalities over uniform declarations. We show this by establishing NP-completeness of the complement problem: checking type disequalities.

▶ **Theorem 16.** *Deciding type disequalities over uniform declarations is NP-complete.*

Proof. The sketch is as follows: Satisfiabiliy of a 3-CNF formula φ can be encoded as the type disequality $\tau(\varphi) \not\approx \bot$. This establishes NP-hardness, but it remains to be shown that if $\tau \not\approx \tau'$ then there is a polynomial size witness. This witness is shown to exist due to the structure of uniform declarations. ◀

5 Canonical Forms

In this section we develop a process for *canonizing* type expressions over uniform declarations. The type canonizer reduces the problem of deciding type equations to checking syntactic equality of canonical forms. To simplify theorems, we assume non-auxiliary variables never denote the empty set: $[\![\alpha_f]\!] \neq \emptyset$. It should be possible to construct at least one well-typed term for a given data constructor.

▶ **Definition 17** (Type Canonizer). For uniform declarations $\mathcal{D}$, a *type canonizer* $can()$ is a function from type expressions to type expressions satisfying:

$$can(\bot) = \bot \quad \wedge \quad \tau \approx can(\tau) \quad \wedge \quad \tau \approx \tau' \Leftrightarrow can(\tau) = can(\tau'). \tag{8}$$

5.1 Eliminating intersection and auxiliary variables

Our canonizer takes advantage of the fact that intersection and auxiliary variables can be eliminated from type expressions. Given an input to the canonizer τ, then all auxilliary variables can be eliminated from τ by Theorem 15 using the rewrites:

$$\begin{aligned} &\alpha_{aux} \to \bot, && \text{if } \alpha_{aux} \approx \tau_{aux} \notin \mathcal{D}. \\ &\alpha_{aux} \to \tau_{aux}, && \text{if } \alpha_{aux} \approx \tau_{aux} \in \widehat{\mathcal{D}}. \end{aligned} \tag{9}$$

Similarly, the equations from Lemma 9 eliminate intersections between non-variable atoms. In the context of uniform type declarations, we can also eliminate intersections involving non-auxiliary variables by using the equations:

$$\begin{array}{lrl} (\text{product-}\cap) & \alpha_f \cap f(\tau_1', \tau_2') \approx f(\tau_1 \cap \tau_1', \tau_2 \cap \tau_2'), & \alpha_f \approx f(\tau_1, \tau_2) \in \mathcal{D}. \\ (\text{disjoint-}f, g) & \alpha_f \cap \alpha_g \approx \bot, & \text{when } f \text{ and } g \text{ are different.} \\ (\text{disjoint-}f, g) & \alpha_f \cap g(\tau_1, \tau_2) \approx \bot, & \text{when } f \text{ and } g \text{ are different.} \\ (\text{disjoint-}f, \sigma) & \alpha_f \cap \sigma \approx \bot & \\ (\text{disjoint-}f, \beta) & \alpha_f \cap \beta \approx \bot & \end{array}$$

To completely eliminate intersections it remains to apply the set-theoretical properties of union and intersection from Lemma 9: Distribute intersections over unions and apply idempotency to eliminate redundant intersections. Let us call the procedure *simplify*(τ) that applies the elimination steps for intersection. We now have:

▶ **Lemma 18.** *For uniform $\mathcal{D}$ and any τ, then $\tau \approx simplify(\tau)$ and $simplify(\tau)$ contains neither intersections nor auxiliary variables.*

The remaining two subproblems are canonizing repeated unions and recognizing unfoldings of recursive data types. For example,

$$\tau = f(0,1) \cup f(1,0) \cup f(1,1), \quad \tau' = f(1, 0 \cup 1) \cup f(0 \cup 1, 1).$$

Casual inspection reveals that $\tau \approx \tau'$, even though the two terms have quite different forms. One approach to canonization is to expand constructor applications, so τ' is rewritten τ. However, this approach guarantees a combinatorial blow-up in the size of the canonical form; it is also problematic when infinite base types appear as subterms. On the other hand, if τ' should be the canonical form, then the canonizer must compress τ into τ'; it is less obvious how this can be done. The approach we present uses τ' as the canonical form, and does not eagerly expand unions into singleton constructor applications.

5.2 Base Case: Canonizing Depth-0 Terms

We build the canonizer inductively; the induction is over the depth of terms.

▶ **Definition 19** (Depth)**.** The depth of an term τ is the length of the longest sequence of constructor applications.

$$depth(\tau) \doteq \begin{cases} 1 + max(depth(\tau_1), depth(\tau_2)) & \text{if } \tau = f(\tau_1, \tau_2) \\ max(depth(\tau_1), depth(\tau_2)) & \text{if } \tau = \tau_1 \cup \tau_2, \text{ or } \tau = \tau_1 \cap \tau_2 \\ 0 & \text{otherwise.} \end{cases} \tag{10}$$

A depth-0 term is another name for an application-free term.

▶ **Lemma 20.** *For uniform $\mathcal{D}$ and any term τ, then $depth(simplify(\tau)) \leq depth(\tau)$.*

▶ **Definition 21** (Base Rewrite System)**.** Let $\mathcal{R}_{expand}$ be the term rewrite system given by the rule: $\tau \to \beta \cup \tau$, if $[\![\beta]\!] \subseteq [\![\tau]\!]$. Let $\mathcal{R}_{contract}$ be the term rewrite system: $\tau \cup \beta \to \beta$, if $[\![\tau]\!] \subseteq [\![\beta]\!]$. The term $base(\tau)$ is constructed by first applying $\mathcal{R}_{expand}$ exhaustively (modulo associativity and commutativity (AC) of $\cup$) until no new types can be added to τ. Second, $\mathcal{R}_{contract}$ is applied to eliminate subsumed types.

▶ **Lemma 22.** *For uniform $\mathcal{D}$ and simplified, depth-0 terms τ and τ'. If $\tau \approx \tau'$, then $base(\tau) = base(\tau')$ modulo AC of $\cup$.*

Lemma 22 guarantees syntactic equivalence up to ordering of unions. However, we would like to handle this order precisely. Fix an arbitrary ordering $\prec$ on type terms, then a type term τ is *sorted* if τ is of the form $\tau_1 \cup (\tau_2 \cup (\tau_3 \cup (\ldots \cup \tau_n)))$, and $\tau_1 \prec \tau_2 \prec \tau_3 \prec \ldots \prec \tau_n$, and the type terms $\tau_1, \ldots, \tau_n$ are not union expressions. Let *sort* be a function that takes any type term τ and rewrites it into an equivalent but sorted expression $sort(\tau)$.

▶ **Corollary 23** (Depth-0 Canonizer)**.** *Define can_0 as a function from depth-0 terms to depth-0 terms such that: $can_0(\tau) \doteq sort(base(simplify(\tau)))$. Then can_0 is a type canonizer for depth-0 expressions.*

5.3 Induction: Terms of Depth $k > 0$.

The induction step builds a canonizer can_{k+1} for terms with at most depth $k+1$ from a canonizer can_k for terms with at most depth k. When canonizing union expressions, we utilize the lattice-theoretic properties induced by a can_k canonizer:

▶ **Definition 24.** Let $S = \{\sigma_1, \ldots, \sigma_n\}$ be a finite set of constants and $k \geq 0$ then $\Sigma_\tau(S, k)$ is the set of all terms τ such that $depth(\tau) \leq k$ and $cnsts(\tau) \subseteq S$. Note, $cnsts(\tau)$ is the set of constants appearing as subterms of τ.

▶ **Lemma 25** (S, k-Canonical Lattice). *Let S be a finite set of constants, and can_k be a canonizer for expressions with depth at most k. Define*

$$L(S,k) \doteq \langle can_k(\Sigma_\tau(S,k)), \bot, \top, \sqcap, \sqcup \rangle, \text{ where}$$

$$\bot \doteq \bot_{\Sigma_\tau}, \qquad \top \doteq can_k((\bigcup_{\alpha \in V} \alpha) \cup (\bigcup_{\beta \in B} \beta) \cup (\bigcup_{\sigma \in S} \sigma)),$$

$$\tau \sqcap \tau' \doteq can_k(\tau \cap \tau'), \qquad \tau \sqcup \tau' \doteq can_k(\tau \cup \tau').$$

Then, $L(S,k)$ is a finite lattice such that $\tau \sqcap \tau' \approx \tau \cap \tau'$ and $\tau \sqcup \tau' \approx \tau \cup \tau'$.

The effect of constructor applications on canonized terms of depth k can be understood as an f-labeled product lattice:

▶ **Lemma 26** (f, S, k-Canonical Lattice). *Let f be a binary constructor, S a finite set of constants, and can_k a canonizer, define*

$$L(f,S,k) \doteq \langle U, \bot, \top, \sqcap, \sqcup \rangle, \text{ where}$$

$$U \doteq \bot_{\Sigma_\tau} \cup \left\{ f(\tau, \tau') \;\middle|\; \begin{array}{l} \tau \in L(S,k) \ \setminus \ \{\bot_{\Sigma_\tau}\}, \\ \tau' \in L(S,k) \ \setminus \ \{\bot_{\Sigma_\tau}\} \end{array} \right\}.$$

$$\bot \doteq \bot_{\Sigma_\tau}. \quad \top \doteq f(\top_{L(S,k)}, \top_{L(S,k)}).$$
$$f(\tau_1, \tau_2) \sqcap f(\tau_1', \tau_2') \doteq f(\tau_1 \sqcap_{L(S,k)} \tau_1', \tau_2 \sqcap_{L(S,k)} \tau_2').$$
$$f(\tau_1, \tau_2) \sqcup f(\tau_1', \tau_2') \doteq f(\tau_1 \sqcup_{L(S,k)} \tau_1', \tau_2 \sqcup_{L(S,k)} \tau_2').$$

Then, $L(f,S,k)$ is a finite lattice where $\tau \sqcap \tau' \approx \tau \cap \tau'$ and $\tau \cup \tau' <: \tau \sqcup \tau'$.

The elements of $L(f,S,k)$ already denote unique type expressions (otherwise $L(f,S,k)$ would not be a lattice), and have maximum depth $k+1$. However, the join operation may over-approximate the union of two elements: $f(0,0) \sqcup f(1,1) = f(0 \cup 1, 0 \cup 1)$. Thus, $L(f,S,k)$ cannot be immediately used to canonize unions of terms with depth $k+1$. The following lemmas overcome this limitation:

▶ **Theorem 27** (Maximal Decomposition). *Given $\tau = \tau_1 \cup \ldots \cup \tau_n$ such that $\tau_1, \ldots, \tau_n \in L(f,S,k)$, then a maximal decomposition of τ is an element $\tau' \in L(f,S,k)$ such that:*

$$[\![\tau']\!] \subseteq [\![\tau]\!] \quad \wedge \quad \forall \tau'' \in L(f,S,k) \ \tau' \sqsubset \tau'' \Rightarrow [\![\tau'']\!] \not\subseteq [\![\tau]\!]. \tag{11}$$

Let $decs(\tau)$ be a type term that is the union of all maximal decompositions. It is computed by saturating τ w.r.t. the following implications and keeping the $L(f,S,k)$-maximal subterms:

$$f(\tau_1, \tau_2), f(\tau_1', \tau_2') \in \tau \ \Rightarrow \ f(\tau_1 \sqcap \tau_1', \tau_2) \in \tau. \qquad f(\tau_1, \tau_2), f(\tau_1', \tau_2') \in \tau \ \Rightarrow f(\tau_1, \tau_2 \sqcap \tau_2') \in \tau.$$
$$f(\tau_1, \tau_2), f(\tau_1, \tau_3) \in \tau \ \Rightarrow f(\tau_1, \tau_2 \sqcup \tau_3) \in \tau. \qquad f(\tau_2, \tau_1), f(\tau_3, \tau_1) \in \tau \ \Rightarrow f(\tau_2 \sqcup \tau_3, \tau_1) \in \tau.$$

▶ **Lemma 28** (Correctness\Uniqueness of Decompositions)**.** *Modulo AC of* $\cup$*:*

$$\tau \approx \mathit{decs}(\tau) \quad \wedge \quad \tau \approx \tau' \;\Leftrightarrow\; \mathit{decs}(\tau) = \mathit{decs}(\tau')$$

The union of all maximal decompositions has the same denotation as the original type expression, but is unique for a lattice $L(f, S, k)$. We now have almost all the ingredients to define a complete canonizer. The final task is to ensure that recursive data types are folded and unfolded consistently. Suppose τ is an expression of depth at most $k+1$ and has the form $f(\tau_1', \tau_1'') \cup \ldots \cup f(\tau_n', \tau_n'')$, where every τ_i', τ_i'' is canonical with respect to can_k. We can use a maximal decomposition to canonize this union by creating the type expression $\mathit{sort}(\mathit{fold}(\mathit{decs}(\tau)))$ where *fold* replaces constructor applications with non-auxiliary type variables. The *unfold*() operation expands type variables with their declarations:

$$\begin{array}{lll}
\mathit{fold}(f(\tau_1,\tau_2)) \doteq & \alpha_f, & \alpha_f \approx f(\tau_1^f, \tau_2^f) \in \widehat{\mathcal{D}}, \tau_i = \mathit{can}_k(\tau_i^f), i = 1, 2. \\
\mathit{fold}(\tau_1 \cup \tau_2) \doteq & \mathit{fold}(\tau_1) \cup \mathit{fold}(\tau_2). & \\
\mathit{fold}(\tau) \doteq & \tau, & \text{otherwise.} \\
\mathit{unfold}(\alpha_f) \doteq & f(\mathit{can}_k(\tau_1^f), \mathit{can}_k(\tau_2^f)), & \alpha_f \approx f(\tau_1^f, \tau_2^f) \in \widehat{\mathcal{D}}. \\
\mathit{unfold}(f(\tau_1,\tau_2)) \doteq & f(\mathit{can}_k(\tau_1), \mathit{can}_k(\tau_2)), & \mathit{can}_k(\tau_1) \neq \bot \wedge \mathit{can}_k(\tau_2) \neq \bot. \\
\mathit{unfold}(f(\tau_1,\tau_2)) \doteq & \bot, & \mathit{can}_k(\tau_1) = \bot \vee \mathit{can}_k(\tau_2) = \bot. \\
\mathit{unfold}(\tau_1 \cup \tau_2) \doteq & \mathit{unfold}(\tau_1) \cup \mathit{unfold}(\tau_2). &
\end{array}$$

Then define the canonizer for such terms to be $\mathit{can}_{f,k+1}(\tau) \doteq \mathit{sort}(\mathit{fold}(\mathit{decs}(\mathit{unfold}(\tau))))$.

▶ **Lemma 29** (Canonizer for Unions of f-terms)**.** *Let τ be a sequence of unions of either the type variable α_f or an f-term with depth $\leq k+1$. Also, suppose $\alpha_f \approx f(\tau_1^f, \tau_2^f) \in \widehat{\mathcal{D}}$ and $S = \mathit{cnsts}(\tau) \cup \mathit{cnsts}(\tau_1^f) \cup \mathit{cnsts}(\tau_2^f)$. Then, $\mathit{can}_{f,k+1}$ is a canonizer for terms in $\Sigma_\tau(S, k+1)$.*

The full canonizer is a obtained by canonizing base types together with constants, and then canonizing for each constructor in the signature: If *simplify*(τ) has the form:

$$\underbrace{\alpha_f \cup f(\tau_1, \tau_2)..}_{\tau_f} \cup \underbrace{g(\tau_3, \tau_4) \cup g(\tau_5, \tau_6)}_{\tau_g} \cup \underbrace{\sigma_1 \cup \sigma_2 \ldots \cup \beta_1 \cup \beta_2..}_{\tau_c}$$

▶ **Theorem 30** (Canonizer can_{k+1})**.** *Assume can_k is a canonizer for terms of depth at most k, then can_{k+1} is a canonizer for terms with depth at most $k+1$, where:*

$$\mathit{can}_{k+1}(\tau) \doteq \mathit{sort}(\mathit{base}(\mathit{can}_{f,k+1}(\tau_f) \cup \mathit{can}_{g,k+1}(\tau_g) \cup \tau_c)) \tag{12}$$

6 Conclusion

We developed a type canonizer for regular types expressed as type terms over uniform declarations. The canonizer solves type checking problems and returns type judgments as canonical type terms. We implemented a type system using this approach in the LP language FORMULA [11], and experimentally showed that canonization times behave linearly while canonical forms are of high quality. Please see the full technical report at `http://research.microsoft.com/~ejackson` for experimental data. Future work includes studying the interaction of regular types and constraints: It is well-known that Presburger constraints describe regular sets, so constraints, such as $x = 2y$, can be used to infer that x is even.

References

1 Alexander Aiken and Brian R. Murphy. Implementing Regular Tree Expressions. In *FPCA 1991*, pages 427–447. Springer-Verlag, 1991.

2 Hassan Aït-Kaci and Andreas Podelski. Towards a Meaning of LIFE. *J. Log. Program.*, 16(3):195–234, 1993.

3 Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In Colin Runciman and Olin Shivers, editors, *ICFP 2003*, pages 51–63. ACM, 2003.

4 Lei Chen and Haiming Chen. Subtyping Algorithm of Regular Tree Grammars with Disjoint Production Rules. In *ICTAC 2010*, pages 45–59, 2010.

5 Philip W. Dart and Justin Zobel. A Regular Type Language for Logic Programs. In *Types in Logic Programming*, pages 157–187. MIT Press, 1992.

6 Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM*, 55(4), 2008.

7 T. Fruhwirth, E. Shapiro, M.Y. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *LICS 1991*, pages 300 –309, July 1991.

8 John P. Gallagher and Germán Puebla. Abstract Interpretation over Non-deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *PADL 2002*, pages 243–261, 2002.

9 Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Sci. Comput. Program.*, 58(1-2):115–140, 2005.

10 Haruo Hosoya, Jerome Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

11 Ethan K. Jackson, Eunsuk Kang, Markus Dahlweid, Dirk Seifert, and Thomas Santen. Components, platforms and possibilities: towards generic automation for MDA. In *EMSOFT 2010*, pages 39–48, 2010.

12 David Jeffery, Fergus Henderson, and Zoltan Somogyi. Type Classes in Mercury. In *ACSC 2000*, pages 128–135, 2000.

13 Claudio Vaucheret and Francisco Bueno. More Precise Yet Efficient Type Inference for Logic Programs. In *SAS 2002*, pages 102–116, 2002.