Type Inference by Coinductive Logic Programming*

Davide Ancona, Giovanni Lagorio, and Elena Zucca

DISI, Univ. of Genova, v. Dodecaneso 35, 16146 Genova, Italy
{davide,lagorio,zucca}@disi.unige.it

Abstract. We propose a novel approach to constraint-based type inference based on coinductive logic. Constraint generation corresponds to translation into a conjunction of Horn clauses P, and constraint satisfaction is defined in terms of the coinductive Herbrand model of P. We illustrate the approach by formally defining this translation for a small object-oriented language similar to Featherweight Java, where type annotations in field and method declarations can be omitted.

In this way, we obtain a very precise type inference and provide new insights into the challenging problem of type inference for object-oriented programs. Since the approach is deliberately declarative, we define in fact a formal specification for a general class of algorithms, which can be a useful road map to researchers.

Furthermore, despite we consider here a particular language, the methodology could be used in general for providing abstract specifications of type inference for different kinds of programming languages.

Keywords: Type inference, coinduction, nominal and structural typing, object-oriented languages.

1 Introduction

Type inference is a valuable method to ensure static guarantees on the execution of programs (like the absence of some type errors) and to allow sophisticated compiler optimizations. In the context of object-oriented programming, many solutions have been proposed to perform type analysis (we refer to the recent article of Wang and Smith [20] for a comprehensive overview), but the increasing interest in dynamic object-oriented languages is asking for even more precise and efficient type inference algorithms [3,14].

Two important features which have to be supported by type inference are *parametric* and *data polymorphism* [1]; the former allows invocation of a method on arguments of unrelated types, the latter allows assignment of values of unrelated types to a field. While most solutions proposed in literature support well parametric polymorphism, only few inference algorithms are able to deal properly with data polymorphism; such algorithms, however, turn out to be quite complex and cannot be easily described.

* This work has been partially supported by MIUR EOS DUE - Extensible Object Systems for Dynamic and Unpredictable Environments.

S. Berardi, F. Damiani, and U. de'Liguoro (Eds.): TYPES 2008, LNCS 5497, pp. 1–18, 2009.

In this paper we propose a novel approach to type inference, by exploiting coinductive logic programming. Our approach is deliberately declarative, that is, we do not define any algorithm, but rather try to capture a space of possible solutions to the challenging problem of precise type inference of object-oriented programs.

The basic idea is that the program to be analyzed can be translated into an approximating logic program and a goal; then, type inference corresponds to find an instantiation of the goal which belongs to the coinductive model of the logic program. Coinduction allows to deal in a natural way with both recursive types [11,12] and mutually recursive methods.

The approach is fully formalized for a purely functional object-oriented language similar to Featherweight Java [16], where type annotations can be omitted, and are used by the programmer only as subtyping constraints. The resulting type inference is very powerful and allows, for instance, very precise analysis of heterogeneous container objects (as linked lists).

The paper is structured as follows: Section 2 defines the language and gives an informal presentation of the type system, based on standard recursive and union types. In Section 3 the type system is reconsidered in the light of coinductive logic programming, and the translation is fully formalized. Type soundness w.r.t. the operational semantics is claimed (proofs are sketched in Appendix B). Finally, Section 4 draws some conclusions and discusses future developments.

2 Language Definition and Types

In this section we present a simple object-oriented (shortly OO) language together with the definition of types. Constraint generation and satisfaction are only informally illustrated; they will be formally defined in the next section, on top of coinductive logic programming.

2.1 Syntax and Operational Semantics

The syntax is given in Figure 1. Syntactic assumptions listed in the figure are verified before performing type inference. We use bars for denoting sequences: for instance, $\overline{e}^m$ denotes $e_1, \ldots, e_m$, $\overline{T\ x}^n$ denotes[1] $T_1\ x_1, \ldots, T_n\ x_n$, and so on.

The language is basically Featherweight Java (FJ) [16], a small Java subset which has become a standard example to illustrate extensions and new technologies for Java-like languages. Since we are interested in type inference, type annotations for parameters, fields, and returned values can be omitted; furthermore, to make the type inference problem more interesting, we have introduced the conditional expression `if` (e) e_1 `else` e_2, and a more expressive form of constructor declaration.

We assume countably infinite sets of *class names* c, *method names* m, *field names* f, and *parameter names* x. A program is a sequence of class declarations

[1] If not explicitly stated, the bar "distributes over" all meta-variables below it.

$$
\begin{array}{rcl}
\mathit{prog} & ::= & \overline{cd}^{\,n}\ e \\
\mathit{cd} & ::= & \texttt{class}\ c_1\ \texttt{extends}\ c_2\ \{\ \overline{fd}^{\,n}\ \ cn\ \ \overline{md}^{\,m}\ \}\quad (c_1 \neq \texttt{Object}) \\
\mathit{fd} & ::= & T\ f; \\
\mathit{cn} & ::= & c(\overline{T\ x}^{\,n})\ \{\texttt{super}(\overline{e}^{\,m}); \overline{f = e'; }^{\,k}\} \\
\mathit{md} & ::= & T_0\ m(\overline{T\ x}^{\,n})\ \{e\} \\
e & ::= & \texttt{new}\ c(\overline{e}^{\,n}) \mid x \mid e.f \mid e_0.m(\overline{e}^{\,n}) \mid \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2 \mid \texttt{false} \mid \texttt{true} \\
T & ::= & N \mid \epsilon \\
N & ::= & c \mid \texttt{bool} \\
v & ::= & \texttt{new}\ c(\overline{v}^{\,n}) \mid \texttt{false} \mid \texttt{true}
\end{array}
$$

Assumptions: $n, m, k \geq 0$, inheritance is not cyclic, names of declared classes in a program, methods and fields in a class, and parameters in a method are distinct.

Fig. 1. Syntax of OO programs

together with a main expression from which the computation starts. A class declaration consists of the name of the declared class and of its direct superclass (hence, only single inheritance is supported), a sequence of field declarations, a constructor declaration, and a sequence of method declarations. We assume a predefined class `Object`, which is the root of the inheritance tree and contains no fields, no methods and a constructor with no parameters. A field declaration consists of a type annotation and a field name. A constructor declaration consists of the name of the class where the constructor is declared, a sequence of parameters with their type annotations, and the body, which consists of an invocation of the superclass constructor and a sequence of field initializations, one for each field declared in the class.[2] A method declaration consists of a return type annotation, a method name, a sequence of parameters with their type annotations, and an expression (the method body).

Expressions are standard; boolean values and conditional expressions have been introduced just to show how the type system allows precise typing in case of branches. Integer values and the related standard primitives will be used in the examples, but are omitted in the formalization, since their introduction would only imply a straightforward extension of the type system. As in FJ, *this* is considered as a special implicit parameter.

A type annotation T can be either a nominal type N (the primitive type `bool` or a class name c) or empty.

Finally, the definition of values v is instrumental to the (standard) small steps operational semantics of the language, indexed over the class declarations defined by the program, shown in Figure 2.

For reasons of space, side conditions have been placed together with premises, and standard contextual closure have been omitted. To be as general as possible, no evaluation strategy has been fixed. Auxiliary functions *cbody* and *mbody* are defined in Appendix A.

[2] This is a generalization of constructors of FJ, whose arguments exactly match in number and type the fields of the class, and are used as initialization expressions.

$$\text{(field-1)}\ \frac{cbody(cds, c) = (\overline{x}^n, \{\texttt{super}(\ldots); \overline{f = e'; }^k\}) \qquad f = f_i \qquad 1 \le i \le k}{\texttt{new}\ c(\overline{e}^n).f \rightarrow_{cds} e'_i[\overline{e}^n/\overline{x}^n]}$$

$$\text{(field-2)}\ \frac{\begin{array}{l} cbody(cds, c) = (\overline{x}^n, \{\texttt{super}(\overline{e'}^m); \overline{f = \ldots; }^k\}) \\ \forall i \in 1..k \quad f \neq f_i \qquad \texttt{class}\ c\ \texttt{extends}\ c'\ \{\ \ldots\} \in cds \\ \texttt{new}\ c'(e'_1[\overline{e}^n/\overline{x}^n], \ldots, e'_m[\overline{e}^n/\overline{x}^n]).f \rightarrow_{cds} e \end{array}}{\texttt{new}\ c(\overline{e}^n).f \rightarrow_{cds} e}$$

$$\text{(invk)}\ \frac{mbody(cds, c, m) = (\overline{x}^n, e) \qquad e_{this} = \texttt{new}\ c(\overline{e}^k)}{\texttt{new}\ c(\overline{e}^k).m(\overline{e'}^n) \rightarrow_{cds} e[\overline{e'}^n/\overline{x}^n][e_{this}/this]}$$

$$\text{(if-1)}\ \frac{}{\texttt{if}\ (\texttt{true})\ e_1\ \texttt{else}\ e_2 \rightarrow_{cds} e_1} \qquad \text{(if-2)}\ \frac{}{\texttt{if}\ (\texttt{false})\ e_1\ \texttt{else}\ e_2 \rightarrow_{cds} e_2}$$

Fig. 2. Reduction rules for OO programs

Rule (field-1) corresponds to the case where the field f is declared in the same class of the constructor, whereas rule (field-2) covers the disjoint case where the field has been declared in some superclass. The notation $e[\overline{e}^n/\overline{x}^n]$ denotes parallel substitution of x_i by e_i (for $i = 1..n$) in expression e.

In rule (invk), the parameters and the body of the method to be invoked are retrieved by the auxiliary function *mbody*, which performs the standard method look-up. If the method is found, then the invocation reduces to the body of the method where the parameters are substituted by the corresponding arguments, and *this* by the receiver object (the object on which the method is invoked).

The remaining rules are trivial.

The one step reduction relation on programs is defined by: $(cds\ e) \rightarrow (cds\ e')$ iff $e \rightarrow_{cds} e'$. Finally, $\rightarrow^*$ and $\rightarrow^*_{cds}$ denote the reflexive and transitive closures of $\rightarrow$ and $\rightarrow_{cds}$, respectively.

2.2 Types

Types, class environments and constraints are defined in Figure 3.

Value types (meta-variable τ) must not be confused with nominal types (meta-variable N) in the OO syntax. Nominal types are used as type annotations by

$$\begin{array}{rl} \tau & ::= X \mid bool \mid obj(c, \rho) \mid \tau_1 \vee \tau_2 \mid \mu X.\tau \quad (\mu X.\tau \text{ contractive}) \\ \rho & ::= [\overline{f{:}\tau}^n] \\ \Delta & ::= \overline{c{:}(c', fts, ct, mts)}^n \\ fts & ::= [\overline{f{:}T}^n] \\ ct & ::= \forall \overline{X}^n.C \Rightarrow ((\prod_{i=1..k} X'_i) \rightarrow obj(c, \rho)) \quad (\{\overline{X'}^k\} \subseteq \{\overline{X}^n\}) \\ mts & ::= [\overline{m{:}mt}^n] \\ mt & ::= \forall \overline{X}^n.C \Rightarrow ((\prod_{i=1..k} X'_i) \rightarrow \tau) \quad (\{\overline{X'}^k\} \subseteq \{\overline{X}^n\}, n \ge k \ge 1) \\ C & ::= \{\overline{\gamma}^n\} \\ \gamma & ::= inst\_of(\tau, N) \mid new(c, [\overline{\tau}^n], \tau) \mid fld\_acc(\tau_1, f, \tau_2) \\ & \quad \mid invk(\tau_0, m, [\overline{\tau}^n], \tau) \mid cond(\tau_1, \tau_2, \tau_3, \tau) \end{array}$$

Fig. 3. Definition of types, class environments and constraints

programmers, whereas value types are used in the type system and are transparent to programmers. Nominal types are approximations[3] of the much more precise value types. This is formally captured by the constraint $inst\_of(\tau, N)$ (see in the following).

A value type can be a type variable X, the primitive type *bool*, an object type $obj(c, \rho)$, a union type $\tau_1 \vee \tau_2$, or a recursive type $\mu X.\tau$.

An object type $obj(c, \rho)$ consists of the class c of the object and of a record type $\rho = [\overline{f{:}\tau}^n]$ specifying the types of the fields. Field types need to be associated with each object, to support data polymorphism; the types of methods can be retrieved from the class c of the object (see the notion of class environment below).

Union types [10,15] have the conventional meaning: an expression of type $\tau_1 \vee \tau_2$ is expected to assume values of type τ_1 or τ_2.

Recursive types are standard [2]: intuitively, $\mu X.\tau$ denotes the recursive type defined by the equation $X = \tau$, thus fulfilling the equivalences $\mu X.\tau \equiv \tau[\mu X.\tau/X]$ and $\mu X.\tau \equiv \mu X'.\tau[X'/X]$, where substitutions are capture avoiding. As usual, to rule out recursive types whose equation has no unique solution[4], we consider only *contractive* types [2]: $\mu X.\tau$ is contractive iff (1) all free occurrences of X in τ appear inside an object type $obj(c, \rho)$, (2) all recursive types in τ are contractive.

A class environment Δ is a finite map associating with each defined class name c all its relevant type information: the direct superclass; the type annotations associated with each declared field (*fts*); the type of the constructor (*ct*); the type of each declared method (*mts*).

Constructor types can be seen as particular method types. The method type $\forall \overline{X}^n.C \Rightarrow ((\prod_{i=1..k} X'_i) \to \tau)$ is read as follows: for all type variables $\overline{X}^n$, if the finite set of constraints C is satisfied, then the type of the method is a function from $\prod_{i=1..k} X'_i$ to τ. Without any loss of generality, we assume distinct type variables for the parameters; furthermore, the first type variable corresponds to the special implicit parameter *this*, therefore the type $\forall \overline{X}^n.C \Rightarrow ((\prod_{i=1..k} X'_i) \to \tau)$ corresponds to a method with $k-1$ parameters. Finally, note that C and τ may contain other universally quantified type variables (hence, $\{\overline{X'}^k\}$ is a subset of $\{\overline{X}^n\}$).

Constructor types correspond to functions which always return an object type and do not have the implicit parameter *this* (hence, k corresponds to the number of parameters).

Constraints are based on our long-term experience on compositional type-checking and type inference of Java-like languages [6,9,5,17,7]. Each kind of compound expression comes with a specific constraint:

- $new(c, [\overline{\tau}^n], \tau)$ corresponds to object creation, c is the class of the invoked constructor, $\overline{\tau}^n$ the types of the arguments, and τ the type of the newly created object;
- $fld\_acc(\tau_1, f, \tau_2)$ corresponds to field access, τ_1 is the type of the receiver, f the field name, and τ_2 the resulting type of the whole expression;

[3] Except for the type *bool*.

[4] For instance, $\mu X.X$ or $\mu X.X \vee X$.

- $invk(\tau_0, m, [\overline{\tau}^n], \tau)$ corresponds to method invocation, τ_0 is the type of the receiver, m the method name, $\overline{\tau}^n$ the types of the arguments, and τ the type of the returned value;
- $cond(\tau_1, \tau_2, \tau_3, \tau)$ corresponds to conditional expression[5], τ_1 is the type of the condition, τ_2 and τ_3 the types of the "then" and "else" branches, respectively, and τ the resulting type of the whole expression.

The constraint $inst\_of(\tau, N)$ does not correspond to any kind of expression, but is needed for checking that value type τ is approximated by nominal type N.

As it is customary, in the constraint-based approach type inference is performed in two distinct steps: constraint generation, and constraint satisfaction.

Constraint Generation. Constraint generation is the easiest part of type inference. A program *cds* e is translated into a pair (Δ, C), where Δ is obtained from *cds*, and C from e. As we will formally define in the next section, Δ can be represented by a set of Horn clauses, and C by a goal. To give an intuition, consider the following method declaration:

```
class List extends Object {
   altList(i,x){
      if(i<=0) new EList()
      else new NEList(x,this.altList(i-1,x.succ()))
   }
}
```

The method type of `altList` is inferred by collecting all constraints generated from its body:

$$\forall This, I, X, R_1, R_2, R_3, R_4, R_5.$$
$$\left\{ \begin{array}{l} inst\_of(This, List), inst\_of(I, int), new(EList, [\,], R_1), invk(X, succ, [\,], R_2), \\ invk(This, altList, [int, R_2], R_3), new(NEList, [X, R_3], R_4), cond(bool, R_1, R_4, R_5) \end{array} \right\}$$
$$\Rightarrow ((This \times I \times X) \rightarrow R_5)$$

For simplicity we have simplified the set of constraints, omitting the constraints of `i<=0` and `i-1`. The constraint $inst\_of(This, List)$ forces the receiver object to be an instance of (a subclass of) `List`, since the method is declared in class `List`. The other constraints derive from each compound subexpression in the body of the method.

Constraint Satisfaction. After generating the pair (Δ, C) from the program *cds* e, to ensure that the execution of *cds* e is type-safe, one needs to prove that the set of constraints C is satisfiable in the class environment Δ. Typically, in constraint-based type inference of object-oriented programs, constraint satisfaction is defined operationally: most approaches directly provide an algorithm, or, at their best, a framework which can be instantiated by various algorithms

[5] This constraint could be easily avoided in practice, but has been introduced to show how a general methodology can be adopted, by associating with each kind of compound expression a specific constraint.

[20], but a declarative definition of constraint satisfaction is often missing. Even though this operational approach guarantees that type inference is decidable, providing a declarative definition of satisfiability based on a logical model allows one to abstract away from any possible implementation, and to give a simpler specification of the underlying type system. In this paper we take the opposite approach, by defining constraint satisfaction in terms of coinductive logic. In this way, we obtain a very powerful type system which, in fact, is not decidable, but can be approximated by precise type inference algorithms [8,4].

In the last part of this section we provide just an example to show how coinductive logic supports very precise typing. Let us add to the class `List` above the following class declarations:

```
class EList extends List {
   EList(){super();}

}

class A extends Object {
   A(){super();}
   succ(){new B()}
}
```

```
class NEList extends List {
   el; next;
   NEList(e,n){super();
               el=e;next=n;}}

class B extends Object {
   B(){super();}
   succ(){new A()}
}
```

In such a program, the main expression **new** `List().altlist(`i`,new A())` returns an empty list if $i \leq 0$; otherwise, a non empty list is returned whose length is i and whose elements are alternating instances of class `A` and `B` (starting from an `A` instance). Similarly, **new** `List().altlist(`i`,new B())` returns an alternating list starting with a `B` instance.

The results of these two expressions can be specified by the following two precise types, respectively:

$$\begin{aligned}\tau_A = \mu X.\, &obj(EList, [\,]) \vee \\ &obj(NEList, [el{:}obj(A, [\,]), next{:}\, obj(EList, [\,]) \vee \\ &\qquad obj(NEList, [el{:}obj(B, [\,]), next{:}X])])\\ \tau_B = \mu X.\, &obj(EList, [\,]) \vee \\ &obj(NEList, [el{:}obj(B, [\,]), next{:}\, obj(EList, [\,]) \vee \\ &\qquad obj(NEList, [el{:}obj(A, [\,]), next{:}X])])\end{aligned}$$

By unfolding and coinduction, the following two type equivalences hold:

$$\begin{aligned}\tau_A &\equiv obj(EList, [\,]) \vee obj(NEList, [el{:}obj(A, [\,]), next{:}\tau_B])\\ \tau_B &\equiv obj(EList, [\,]) \vee obj(NEList, [el{:}obj(B, [\,]), next{:}\tau_A])\end{aligned}$$

We show now that in the class environment corresponding to the example program, the constraints

$$\begin{aligned}&invk(obj(List, [\,]), altList, [int, obj(A, [\,])], X_A)\\ &invk(obj(List, [\,]), altList, [int, obj(B, [\,])], X_B)\end{aligned}$$

generated from the two expressions are satisfiable for $X_A = \tau_A$ and $X_B = \tau_B$. For the first constraint we have to prove that the constraints of the method type

of `altList` are satisfiable for $This = obj(List, [\,])$, $I = int$, and $X = obj(A, [\,])$. That is, the following set is satisfiable.

$$\left\{ \begin{array}{l} inst\_of(obj(List, [\,]), List), inst\_of(int, int), new(EList, [\,], R_1), \\ invk(obj(A, [\,]), succ, [\,], R_2), invk(obj(List, [\,]), altList, [int, R_2], R_3), \\ new(NEList, [obj(A, [\,]), R_3], R_4), cond(bool, R_1, R_4, R_5) \end{array} \right\}$$

The two *inst_of* constraints are trivially satisfied, whereas $new(EList, [\,], R_1)$ and $invk(obj(A, [\,]), succ, [\,], R_2)$ are satisfiable for $R_1 = obj(EList, [\,])$ and $R_2 = obj(B, [\,])$. Then, by coinduction, $invk(obj(List, [\,]), altList, [int, R_2], R_3)$ is satisfiable for $R_3 = \tau_B$. Consequently, $new(NEList, [obj(A, [\,]), R_3], R_4)$ is satisfiable for $R_4 = obj(NEList, [el{:}obj(A, [\,]), next{:}\tau_B])$, and $cond(bool, R_1, R_4, R_5)$ for $R_5 = obj(EList, [\,]) \vee obj(NEList, [el{:}obj(A, [\,]), next{:}\tau_B]) \equiv \tau_A$. This last equivalence can be proved by unfolding and coinduction. The proof for the other constraint is symmetric.

3 Reconsidered Type Inference System

In this section we reconsider the type inference system described in the previous section in the light of coinductive logic.

The first basic idea consists in representing a class environment as a conjunction of Horn clauses (that is, a logic program), a set of type constraints as a conjunction of atoms (predicates applied to terms), and value types as terms. In this way, constraint generation corresponds to a translation from an OO program *cds e* to a pair (P, B), where P is a logic program corresponding to the class environment generated from *cds*, and B is a conjunction of atoms corresponding to the constraints generated from e.

We assume two countably infinite sets of *predicate* p and *function* f symbols, respectively, each one with an associated *arity* $n \geq 0$, and a countably infinite set of *logical variables* X. Functions with arity 0 are called *constants*. We write p/n, f/n to mean that predicate p, function f have arity n, respectively. For symbols we follow the usual convention: function and predicate symbols always begin with a lowercase letter, whereas variables always begin with an uppercase letter.

A logic program is a finite conjunction of *clauses* of the form $A \leftarrow B$, where A is the *head* and B is the *body*. The head is an *atom*, while the body is a finite and possibly empty conjunction of atoms; the empty conjunction is denoted by *true*. A clause with an empty body (denoted by $A \leftarrow true$) is called a *fact*. An atom has the form[6] $p(\overline{t}^n)$ where the predicate p has arity n and $\overline{t}^n$ are *terms*.

For list terms we use the standard notation $[\,]$ for the empty list and $[\_|\_]$ for the list constructor, and adopt the syntax abbreviation $[\overline{t}^n]$ for $[t_1|[\ldots[t_n|[\,]]]$.

In coinductive Herbrand models, terms are possibly infinite trees. The definition of tree which follows is quite standard [13,2]. A path p is a finite and

[6] Parentheses are omitted for predicate symbols of arity 0; the same convention applies for function applications, see below.

possibly empty sequence of natural numbers. The empty path is denoted by ϵ, $p_1 \cdot p_2$ denotes the concatenation of p_1 and p_2, and $|p|$ denotes the length of p. A tree t is a partial function from paths to logical variables and function symbols, satisfying the following conditions:

1. the domain of t (denoted by $dom(t)$) is prefix-closed and not empty;
2. for all paths p in $dom(t)$ and for all natural numbers n,

$$p \cdot n \in dom(t) \text{ iff } t(p) = f/m \text{ and } n < m.$$

If $p \in dom(t)$, then the subtree t' of t rooted at p is defined by $dom(t') = \{p' \mid p \cdot p' \in dom(t)\}$, $t'(p') = t(p \cdot p')$; t' is said a *proper* subset of t iff $p \neq \emptyset$.

Note that recursive types defined with μ correspond to *regular* trees (see below), while here we are considering also types corresponding to non regular trees, therefore the set of types is much more expressive than that defined in the previous section, and, in fact, allows much more precise typings [4]. This is perfectly reasonable for a declarative definition of type inference; implementations of the system can only be sound approximations restricted to regular trees. A tree is regular (a.k.a. rational) if and only if it has a finite number of distinct subtrees. Regular terms can be finitely represented by means of term unification problems [19], that is, finite sets of equations [13,2] of the form $X = t$ (where t is a finite term which is not a variable). Note that logic programs are built over finite terms; infinite terms are only needed for defining coinductive Herbrand models [19] (co-Herbrand models for short, see Section 3.4).

3.1 Restricted Co-herbrand Universe

Given an OO program *prog*, the co-Herbrand universe [19] of its logic counterpart is the set of all terms built on [ ], *bool*, all constant symbols corresponding to class, field, and method names declared in *prog*, and the symbols of arity 2 [_|_], _ : _, *obj*, and _ ∨ _.

The co-Herbrand universe contains also terms which are non contractive types, as that defined by $X = X \vee X$. The definition of contractive type given in Section 2 can be generalized in a natural way to non regular terms as follows. A term t is contractive iff there exists no countable infinite sequence of natural numbers s s.t. there exists n s.t. for all paths p which are prefixes[7] of s, if $|p| \geq n$, then $p \in dom(t)$, and $t(p) = \vee/2$.

3.2 Restricted Co-herbrand Base

Given an OO program *prog*, the restricted co-Herbrand base of its logical encoding is the set of all ground atoms built on the contractive terms of the restricted co-Herbrand universe and on the following predicate symbols:

– all symbols of the type constraints defined in Figure 3 with the corresponding arity: *inst_of*/2, *new*/3, *fld_acc*/3, *invk*/4, *cond*/4;

[7] Recall that paths are finite sequences.

- *class*/1, where $\mathit{class}(c)$ means that c is a defined class;
- *ext*/2, where $\mathit{ext}(c_1, c_2)$ means that c_1 extends c_2;
- *subclass*/2, where $\mathit{subclass}(c_1, c_2)$ means that c_1 is equal to or is a subclass of c_2;
- *has_fld*/3, where $\mathit{has\_fld}(c, f, T)$ means that class c has field f with type annotation T;
- *fld*/3, where $\mathit{fld}(\rho, f, \tau)$ means that the record type ρ has field f of type τ;
- *dec_fld*/3, where $\mathit{dec\_fld}(c, f, T)$ means that class c contains the declaration of field f with type annotation T;
- *dec_meth*/2 where $\mathit{dec\_meth}(c, m)$ means that c contains the declaration of method m;
- *meth*/4 where $\mathit{meth}(c, m, [\tau_0, \overline{\tau}^n], \tau)$ means that class c has a method m which returns a value of type τ when invoked on receiver of type τ_0 and with arguments of types $\overline{\tau}^n$.

These predicates are needed for translating class environments in logic programs (see Figure 4).

3.3 Constraint Generation

Constraint generation is defined in Figure 4. For the translation we assume bijections from the three sets of class, field and method names declared in the program to three corresponding sets containing constants of the co-Herbrand universe, and bijections from the two sets of parameter names and type variables to two corresponding sets containing logical variables. Given a class name c, a field name f, a method name m, a parameter name x, and a type variable X, we denote with $\widehat{c}$, $\widehat{f}$, $\widehat{m}$ the corresponding constants in the co-Herbrand universe, and with $\widehat{x}$ and $\widehat{X}$ the corresponding logical variables. For simplicity, we assume that the implicit parameter *this* is mapped to the logical variable *This* ($\widehat{\mathit{this}} = \mathit{This}$).

The rules define a judgment for each syntactic category of the OO language:

- $\mathit{prog} \rightsquigarrow (P, B)$: a program is translated in a pair where the first component is a logic program, and the second is a conjunction of atoms which is satisfiable in P iff *prog* is well-typed (see Section 3.4);
- fds **in** $c \rightsquigarrow \mathit{Cl}$, mds **in** $c \rightsquigarrow P$: a field declaration is translated in a clause, whereas a method declaration is translated in a logic program (consisting of two clauses); both kinds of translation depend on the name of the class where the declaration is contained;
- cn **in** $\mathit{fds} \rightsquigarrow \mathit{Cl}$: a constructor declaration is translated in a clause and is defined only if all fields in *fds* are initialized by the constructor in the same order[8] as they are declared in *fds*;
- e **in** $V \rightsquigarrow (t \mid B)$: an expression is translated in a pair where the first component is a term corresponding to the value type of the expression, and the second is a conjunction of atoms corresponding to the generated constraints. Constraint generation succeeds only if all free variables of e are contained in the set of variables V.

[8] This last restriction is just for simplicity.

$$\text{(prog)}\ \frac{\overline{cd \rightsquigarrow P}^{\,n} \qquad e\ \mathbf{in}\ \emptyset \rightsquigarrow (t \mid B)}{\overline{cd}^{\,n}\ e \rightsquigarrow (P_{default} \cup (\cup_{i=1..n} P_i), B)} \qquad \text{(field)}\ \frac{}{T\ f;\ \mathbf{in}\ c \rightsquigarrow dec\_fld(\widehat{c}, \widehat{f}, \widehat{T}) \leftarrow true.}$$

$$\text{(class)}\ \frac{\overline{fd\ \mathbf{in}\ c_1 \rightsquigarrow P^F}^{\,n} \quad cn\ \mathbf{in}\ \overline{fd}^{\,n} \rightsquigarrow Cl \quad \overline{md\ \mathbf{in}\ c_1 \rightsquigarrow P^M}^{\,m} \quad \begin{array}{l} P^F = \cup_{i=1..n} P_i^F \\ P^M = \cup_{i=1..m} P_i^M \end{array}}{\texttt{class}\ c_1\ \texttt{extends}\ c_2\ \{\ \overline{fd}^{\,n}\ cn\ \overline{md}^{\,m}\ \} \rightsquigarrow \left\{ \begin{array}{l} class(\widehat{c_1}) \leftarrow true. \\ ext(\widehat{c_1}, \widehat{c_2}) \leftarrow true. \end{array} \right\} \cup P^F \cup \{Cl\} \cup P^M}$$

$$\text{(constr-dec)}\ \frac{\overline{e\ \mathbf{in}\ \{\overline{x}^n\} \rightsquigarrow (t \mid B)}^{\,m} \quad \overline{e'\ \mathbf{in}\ \{\overline{x}^n\} \rightsquigarrow (t' \mid B')}^{\,k}}{\begin{array}{l} c(\overline{T\ x}^{\,n})\ \{\texttt{super}(\overline{e}^m); \overline{f = e;}^{\,k}\}\ \mathbf{in}\ \overline{T'\ f;}^{\,k} \rightsquigarrow \\ \quad new(\widehat{c}, [\overline{\widehat{x}}^n], obj(\widehat{c}, [\overline{\widehat{f}:t'}^{\,k} \mid R])) \leftarrow \overline{inst\_of(\widehat{x}, \widehat{T})}^{\,n}, \overline{B}^m, ext(\widehat{c}, C), \\ \qquad\qquad new(C, [\overline{t}^m], obj(C, R)), \overline{B', inst\_of(t', \widehat{T'})}^{\,k}. \end{array}}$$

$$\text{(meth-dec)}\ \frac{e\ \mathbf{in}\ \{This, \overline{\widehat{x}}^n\} \rightsquigarrow (t \mid B)}{\begin{array}{l} T_0\ m(\overline{T\ x}^{\,n})\{e\}\ \mathbf{in}\ c \rightsquigarrow \\ \quad dec\_meth(\widehat{c}, \widehat{m}) \leftarrow true. \\ \quad meth(\widehat{c}, \widehat{m}, [This, \overline{\widehat{x}}^n], t) \leftarrow inst\_of(This, \widehat{c}), \overline{inst\_of(\widehat{x}, \widehat{T})}^{\,n}, B, inst\_of(t, \widehat{T_0}). \end{array}}$$

$$\text{(new)}\ \frac{\overline{e\ \mathbf{in}\ V \rightsquigarrow (t \mid B)}^{\,n}}{\texttt{new}\ c(\overline{e}^n)\ \mathbf{in}\ V \rightsquigarrow (R \mid \overline{B}^n, new(\widehat{c}, [\overline{t}^n], R))}\ R \text{ fresh}$$

$$\text{(var)}\ \frac{}{x\ \mathbf{in}\ V \rightsquigarrow (\widehat{x} \mid true)}\ x \in V \qquad \text{(field-acc)}\ \frac{e\ \mathbf{in}\ V \rightsquigarrow (t \mid B)}{e.f\ \mathbf{in}\ V \rightsquigarrow (R \mid B, fld\_acc(t, \widehat{f}, R))}\ R \text{ fresh}$$

$$\text{(invk)}\ \frac{e_0\ \mathbf{in}\ V \rightsquigarrow (t_0 \mid B_0) \quad \overline{e\ \mathbf{in}\ V \rightsquigarrow (t \mid B)}^{\,n}}{e_0.m(\overline{e}^n)\ \mathbf{in}\ V \rightsquigarrow (R \mid B_0, \overline{B}^n, invk(t_0, \widehat{m}, [\overline{t}^n], R))}\ R \text{ fresh}$$

$$\text{(if)}\ \frac{e\ \mathbf{in}\ V \rightsquigarrow (t \mid B) \quad e_1\ \mathbf{in}\ V \rightsquigarrow (t_1 \mid B_1) \quad e_2\ \mathbf{in}\ V \rightsquigarrow (t_2 \mid B_2)}{\texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2\ \mathbf{in}\ V \rightsquigarrow (R \mid B, B_1, B_2, cond(t, t_1, t_2, R))}\ R \text{ fresh}$$

$$\text{(true)}\ \frac{}{\texttt{true}\ \mathbf{in}\ V \rightsquigarrow (bool \mid true)} \qquad \text{(false)}\ \frac{}{\texttt{false}\ \mathbf{in}\ V \rightsquigarrow (bool \mid true)}$$

Fig. 4. Constraint generation

In rule (class), $\overline{fd\ \mathbf{in}\ c_1 \rightsquigarrow P^F}^{\,n}$ abbreviates $fd_1\ \mathbf{in}\ c_1 \rightsquigarrow P_1^F, \ldots, fd_n\ \mathbf{in}\ c_1 \rightsquigarrow P_n^F$ (the abbreviation $\overline{md\ \mathbf{in}\ c_1 \rightsquigarrow P^M}^{\,m}$ has a similar meaning).

In rule (constr-dec), $e_1\ \mathbf{in}\ \{\overline{x}^n\} \rightsquigarrow (t_1 \mid B_1), \ldots, e_m\ \mathbf{in}\ \{\overline{x}^n\} \rightsquigarrow (t_m \mid B_m)$ is abbreviated by $\overline{e\ \mathbf{in}\ \{\overline{x}^n\} \rightsquigarrow (t \mid B)}^{\,m}$ (the same comment applies to the other premises of the rule).

Most of the rules are self-explanatory; we comment only rules for programs and for constructor and method declarations.

In rule (prog) $P_{default}$ (see Figure 5) contains those clauses shared by any program, whereas $\cup_{i=1..n} P_i$ are the clauses obtained by translating the class declarations of the program. Note that the type t of the main expression e is

$class(object) \leftarrow true.$
$subclass(X, X) \leftarrow class(X).$
$subclass(X, object) \leftarrow class(X).$
$subclass(X, Y) \leftarrow ext(X, Z), subclass(Z, Y).$
$inst\_of(bool, bool) \leftarrow true.$
$inst\_of(obj(C1, X), C2) \leftarrow subclass(C1, C2).$
$inst\_of(T1 \vee T2, C) \leftarrow inst\_of(T1, C), inst\_of(T2, C).$
$fld\_acc(obj(C, R), F, T) \leftarrow has\_fld(C, F, TA), fld(R, F, T), inst\_of(T, TA).$
$fld\_acc(T1 \vee T2, F, FT1 \vee FT2) \leftarrow fld\_acc(T1, F, FT1), fld\_acc(T1, F, FT1).$
$fld([F{:}T|R], F, T) \leftarrow true.$
$fld([F1{:}T1|R], F2, T) \leftarrow fld(R, F2, T), F1 \neq F2.$
$invk(obj(C, S), M, A, R) \leftarrow meth(C, M, [obj(C, S)|A], R).$
$invk(T1 \vee T2, M, A, R1 \vee R2) \leftarrow invk(T1, M, A, R1), invk(T2, M, A, R2).$
$new(object, [\,], obj(object, [\,])) \leftarrow true.$
$has\_fld(C, F, T) \leftarrow dec\_fld(C, F, T).$
$has\_fld(C, F, T1) \leftarrow ext(C, P), has\_fld(P, F, T1), \neg dec\_fld(C, F, T2).$
$meth(C, M, [This|A], R) \leftarrow$
$\quad inst\_of(This, C), ext(C, P), meth(P, M, [This|A], R), \neg dec\_meth(C, M).$
$cond(T1, T2, T3, T2 \vee T3) \leftarrow inst\_of(T1, bool).$

Fig. 5. Clauses in $P_{default}$ shared by all programs

discarded in the consequence of the rule, since only the constraints generated from e are needed to check the type safety of the program.

Note that not all formulas in Figure 5 are Horn clauses; indeed, for brevity we have used the negation of predicates *dec_fld* and *dec_meth*, and the inequality for field names. However, since the set of all field and method names declared in a program is finite, the predicates *not_dec_fld*, *not_dec_meth* and $\neq$ could be trivially defined by conjunctions of facts, therefore all formulas could be turned into Horn clauses.

A constructor declaration generates a single clause whose head has the form $new(\widehat{c}, [\overline{\widehat{x}}^n], obj(\widehat{c}, [\overline{\widehat{f}{:}t'}^k\ |R]))$, where c is the class of the constructor, $\overline{x}^n$ are its parameters, and $obj(\widehat{c}, [\overline{\widehat{f}{:}t'}^k\ |R])$ is the type of the object created by the constructor. This is obviously an object type corresponding to an instance of c, where the types associated with the fields $\overline{f}^k$ declared in c are determined by the initialization expressions $\overline{e'}^k$ (see the second premise), whereas the types associated with the inherited fields are determined by the invocation of the constructor of the direct superclass. Such invocation corresponds to the atom $new(C, [\overline{t}^m], obj(C, R))$; indeed, the atom $ext(\widehat{c}, C)$ is satisfied only if C is instantiated with the direct superclass of c, and the value types $\overline{t}^m$ of the arguments passed to the constructor of C are determined by the expressions $\overline{e}^m$ (see the first premise). Hence, R is the record type associating types with all fields inherited from C. The remaining atoms of the body of the clause are generated either from the expressions $\overline{e}^m$ and $\overline{e'}^k$ (conjunctions of atoms $\overline{B}^m, \overline{B'}^k$), or from the type annotations of the parameters $\overline{x}^n$ and of the fields $\overline{f}^k$ declared in c;

for convenience, we define the translation $\widehat{\epsilon}$ of the empty annotation to always return a fresh variable so that in this case no constraint is actually imposed.

Finally, notice that the clause is correctly generated only if: (1) the free variables of the expressions contained in the constructor body are contained in the set $\{\overline{x}^n\}$ of the parameters (therefore, *this* cannot be accessed); (2) all fields declared in the class are initialized exactly once and in the same order as they are declared.

Rule (meth-dec) is quite similar to (constr-dec) except for: (1) two clauses are generated, one for the predicate *dec_meth* and the other for the predicate *meth*. Notice that *dec_meth* specifies just the names of all methods declared in c, whereas *meth* specifies the names and the types of all methods (either declared or inherited) of c; (2) the variable *this* can be accessed in the body of the method; for this reason, *This* appears as the first parameter in the head of the clause for the predicate *meth*, and *this* is in the set of free variables which can appear in the body e of the method. Obviously, the variable *this* will always contain an instance of (a subclass of) c (see the atom $\mathit{inst\_of}(\mathit{This}, \widehat{c})$).

3.4 Constraint Satisfaction

A substitution θ is a total map from the set of logical variables into the set of contractive terms s.t. $\{X \mid \theta(X) \neq X\}$ is finite. The application of a substitution θ to a term t returns the term $t\theta$ defined as follows:

- $dom(t\theta) =$
  $\{p \mid p \in dom(t)$ or $p = p' \cdot p''$ with $p' \in dom(t)$, $t(p') = X$, and $p'' \in dom(\theta(X))\}$
- $(t\theta)(p) = \begin{cases} t(p) & \text{if } p \in dom(t),\ t(p) \text{ is not a variable} \\ \theta(X)(p'') & \text{if } p = p' \cdot p'',\ p' \in dom(t),\ t(p') = X,\ p'' \in dom(\theta(X)) \end{cases}$

A ground instance of a clause $A \leftarrow \overline{A}^n$ is a ground clause Cl (that is, Cl does not contain logical variables) s.t. $Cl = A\theta \leftarrow \overline{A\theta}^n$ for a substitution[9] θ.

Constraint satisfaction is defined in terms of restricted co-Herbrand models. A restricted co-Herbrand model of a logic program P is a subset of the restricted co-Herbrand base of P which is a fixed-point of the immediate consequence operator T_P from the restricted co-Herbrand base into itself, defined by

$$T_P(S) = \{A \mid A \leftarrow \overline{A}^n \text{ is a ground instance of a clause of } P, \overline{A}^n \in S\}.$$

We have to show that for any program P, T_P is well-defined, that is, is closed w.r.t. contractive terms. This comes from the following proposition.

Proposition 1. *If t is contractive, then $t\theta$ is contractive.*

Proof. See Appendix B. □

Since T_P is obviously monotonic w.r.t. set inclusion, by the Knaster-Tarski theorem there always exists the greatest fixed-point of T_P, which is the greatest restricted co-Herbrand model $M^{co}(P)$ [19] of P.

We say that B is satisfiable in P iff there exists a substitution θ s.t. $B\theta \subseteq M^{co}(P)$.

[9] $\overline{A\theta}^n$ denotes $A_1\theta, \ldots, A_n\theta$.

3.5 Soundness of the System

Soundness follows by progress and subject reduction theorems below; the former states that a well-typed program cannot get stuck, the latter states that if a well-typed program reduces, then it reduces to a well-typed program. The proofs of these two theorems come directly from the main lemmas in Appendix B, whose proofs are a generalization of those which can be found in a companion paper [8].

Theorem 1 (Progress). *If cds* $e \rightsquigarrow (P, B)$ *and* B *is satisfiable in* P*, then either* e *is a value or* $e \rightarrow_{cds} e'$ *for some* e'*.*

Theorem 2 (Subject reduction). *If cds* $e \rightsquigarrow (P, B)$*,* B *is satisfiable in* P*, and* $e \rightarrow_{cds} e'$*, then cds* $e' \rightsquigarrow (P, B')$*, and* B' *is satisfiable in* P*.*

We say that *cds* e is a normal form iff there exists no e' s.t. $(cds\ e) \rightarrow (cds\ e')$. Soundness ensures that reduction of well-typed programs never gets stuck.

Theorem 3 (Soundness). *If cds* $e \rightsquigarrow (P, B)$*,* B *is satisfiable in* P*,* $(cds\ e) \rightarrow^* (cds\ e')$*, and cds* e' *is a normal form, then* e' *is a value.*

Proof. By induction on the number n of reduction steps. The claim for $n = 0$ holds by progress. If $n > 0$, then there exists e'' s.t. $(cds\ e) \rightarrow (cds\ e'')$, and $(cds\ e'') \rightarrow^* (cds\ e')$ in $n - 1$ steps. By subject reduction we have that *cds* $e'' \rightsquigarrow (P, B')$ and B' is satisfiable in P, therefore we can conclude by inductive hypothesis. □

4 Conclusion and Further Developments

We have defined a constraint-based type system for an object-oriented language similar to Featherweight Java, where type annotations in class declarations can be omitted. The type system is specified in a declarative way, by translating programs in sets of Horn clauses and considering their coinductive Herbrand models. This was made possible by our notion of constraints which has been introduced in previous works on principal typing of Java-like languages [9,5].

To our knowledge, this is the first attempt to exploit coinductive logic programming for type inference of object-oriented languages. The resulting type system is very precise and supports well data polymorphism, by allowing precise type inference of heterogeneous container objects (for instance, linked lists containing instances of unrelated classes).

We believe that this approach deserves further developments in several directions.

One of the most interesting and challenging issue concerns the implementation of the type inference defined here in a declarative way. Since the type system is defined on infinite and non regular types, clearly it is not decidable. Nevertheless, devising algorithms restricted to regular types which are sound w.r.t. the type system would represent an important advance in the topic. A possible

implementation can be based on the recent results on the operational semantics of coinductive logic programming [19,18]. We have followed this approach to implement a prototype[10] in Java and Prolog, which is an approximation of the type system able to type the examples presented in this paper. We refer to the companion paper [8] for more details on the implementation.

Scalability and applicability are two other important issues. For the former, it would be interesting to study more complex translations able to deal with flow sensitive analysis and imperative features. To prove that our approach is applicable to other kinds of languages, a first step would consist in defining type inference based on coinductive logic programming for a simple functional language.

References

1. Agesen, O.: The cartesian product algorithm. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 2–26. Springer, Heidelberg (1995)
2. Amadio, R., Cardelli, L.: Subtyping recursive types. ACM Transactions on Programming Languages and Systems 15(4), 575–631 (1993)
3. Ancona, D., Ancona, M., Cuni, A., Matsakis, N.: RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages. In: OOPSLA 2007 Proceedings and Companion, DLS 2007: Proceedings of the 2007 Symposium on Dynamic Languages, pp. 53–64. ACM, New York (2007)
4. Ancona, D., Lagorio, G.: Type systems for object-oriented languages based on coinductive logic. Technical report, DISI - Univ. of Genova (2008) (submitted for publication)
5. Ancona, D., Damiani, F., Drossopoulou, S., Zucca, E.: Polymorphic bytecode: Compositional compilation for Java-like languages. In: ACM Symp. on Principles of Programming Languages 2005, January 2005. ACM Press, New York (2005)
6. Ancona, D., Lagorio, G., Zucca, E.: True separate compilation of Java classes. In: PPDP 2002 - Principles and Practice of Declarative Programming, pp. 189–200. ACM Press, New York (2002)
7. Ancona, D., Lagorio, G., Zucca, E.: Type inference for polymorphic methods in Java-like languages. In: Italiano, G.F., Moggi, E., Laura, L. (eds.) ICTCS 2007 - 10th Italian Conf. on Theoretical Computer Science 2003, eProceedings. World Scientific, Singapore (2007)
8. Ancona, D., Lagorio, G., Zucca, E.: Type inference for Java-like programs by coinductive logic programming. Technical report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova (2008)
9. Ancona, D., Zucca, E.: Principal typings for Java-like languages. In: ACM Symp. on Principles of Programming Languages 2004, pp. 306–317. ACM Press, New York (2004)
10. Barbanera, F., Dezani-Cincaglini, M., de'Liguoro, U.: Intersection and union types: Syntax and semantics. Information and Computation 119(2), 202–230 (1995)
11. Brandt, M., Henglein, F.: Coinductive axiomatization of recursive type equality and subtyping. In: de Groote, P., Hindley, J.R. (eds.) TLCA 1997. LNCS, vol. 1210, pp. 63–81. Springer, Heidelberg (1997)

[10] Available at `http://www.disi.unige.it/person/LagorioG/J2P`

12. Brandt, M., Henglein, F.: Coinductive axiomatization of recursive type equality and subtyping. Fundam. Inform. 33(4), 309–338 (1998)
13. Courcelle, B.: Fundamental properties of infinite trees. Theoretical Computer Science 25, 95–169 (1983)
14. Furr, M., An, J., Foster, J.S., Hicks, M.: Static type inference for Ruby. In: SAC 2009: Proceedings of the 2009 ACM symposium on Applied computing. ACM Press, New York (to appear, 2009)
15. Igarashi, A., Nagira, H.: Union types for object-oriented programming. Journ. of Object Technology 6(2), 47–68 (2007)
16. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems 23(3), 396–450 (2001)
17. Lagorio, G., Zucca, E.: Just: safe unknown types in java-like languages. Journ. of Object Technology 6(2), 69–98 (2007); special issue: OOPS track at SAC 2006
18. Simon, L., Bansal, A., Mallya, A., Gupta, G.: Co-logic programming: Extending logic programming with coinduction. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 472–483. Springer, Heidelberg (2007)
19. Simon, L., Mallya, A., Bansal, A., Gupta, G.: Coinductive logic programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 330–345. Springer, Heidelberg (2006)
20. Wang, T., Smith, S.: Polymorphic constraint-based type inference for objects. Technical report, The Johns Hopkins University (2008) (submitted for publication)

A Auxiliary Functions

$$\text{(mbody-1)}\ \frac{\texttt{class}\ c\ \texttt{extends}\ c'\ \{\ \ldots\ T_0\ m(\overline{T\ x}^n)\{e\}\ldots\}\in cds}{mbody(cds,c,m)=(\overline{x}^n,e)}$$

$$\text{(mbody-2)}\ \frac{\texttt{class}\ c\ \texttt{extends}\ c'\ \{\ \ldots mds\ \}\in cds \qquad m\notin mds \quad mbody(cds,c',m)=(\overline{x}^n,e)}{mbody(cds,c,m)=(\overline{x}^n,e)}$$

$$\text{(cbody)}\ \frac{\texttt{class}\ c\ \texttt{extends}\ c'\ \{\ \ldots c(\overline{T\ x}^n)\ \{\texttt{super}(\overline{e}^m);\overline{f=e';}^k\}\ldots\}\in cds}{cbody(cds,c)=(\overline{x}^n,\{\texttt{super}(\overline{e}^m);\overline{f=e';}^k\})}$$

Fig. 6. Auxiliary functions

B Proofs of Claims of Section 3

Proposition 1. If t is contractive, then $t\theta$ is contractive.

Proof. By contradiction, let us assume that $t\theta$ is not contractive, hence, there exists a countable and infinite sequence s of natural numbers and a natural number n s.t. for all paths p which are prefixes of s if $|p| \geq n$, then $p \in dom(t')$, and $t'(p) = \_\vee\_/2$, for $t' = t\theta$. Let us consider the two following exhaustive and disjoint cases:

- If $p \in dom(t)$ for all paths which are prefixes of s, then t does not contain any variable along p for all finite prefixes p of s, therefore, by definition of $t\theta$, we have $t(p) = t'(p)$ for all finite prefixes p of s, but this contradicts the hypothesis that t is contractive.
- Otherwise, let us consider the longest path p' among all finite prefixes of s s.t. $p' \in dom(t)$, and let $l = |p'|$ (p' exists since we are assuming that there exists a finite prefix of s which does not belong to $dom(t)$, and, by definition of tree, $dom(t)$ is not empty and prefix-closed). Then, by definition of $t\theta$, $p' \in dom(t)$ and $t(p') = X$ for a certain logic variable X, and for all finite prefixes p of s, if $|p| \geq l$, then there exists p'' s.t. $p = p' \cdot p''$, $p'' \in dom(t'')$, and $t''(p'') = t'(p)$, where $t'' = \theta(X)$. Therefore, for all finite prefixes p of s, if $|p| \geq \max(0, n - l)$, then $p \in dom(t'')$, and $t''(p) = \_ \vee \_/2$, which contradicts the hypothesis that $\theta(X)$ is contractive. □

Progress. To prove progress we need the following lemmas.

Lemma 1. *If* $\mathcal{C}[e]$ ***in*** $V \rightsquigarrow (t \mid B)$*, then* e ***in*** $V \rightsquigarrow (t' \mid B')$*, with* $B' \subseteq B$*.*

Proof. By case analysis on the contexts and by induction on their structure. □

Lemma 2. *If cds* $\rightsquigarrow P$*, and* $invk(\widehat{c}, \widehat{m}, [t_1, \ldots, t_n], t)$ *is satisfiable in* P*, then* $mbody(cds, c, m) = (\overline{x}^n, e)$ *for some variables* $\overline{x}^n$ *and expression* e*.*

Proof. By induction on the height of the inheritance tree. □

Theorem 1 [Progress]. If *cds* $e \rightsquigarrow (P, B)$ and B is satisfiable in P, then either e is a value or $e \rightarrow_{cds} e'$ for some e'.

Proof. A generalization of the proof which can be found in a companion paper [8]. □

Subject Reduction. To prove subject reduction we need to introduce a subtyping relation $\leq$ between value types, since after a reduction step the inferred type of the reduced expression may become more specific.

Consider for instance the following expression $e =$ **if true** `1` **else false**. We have $e \rightarrow_{cds} 1$, e **in** $V \rightsquigarrow (X \mid cond(bool, int, bool, X))$ and 1 **in** $V \rightsquigarrow (int \mid true)$. Now $cond(bool, int, bool, X)$ is satisfiable for $X = int \vee bool$, but 1 **in** $V \rightsquigarrow (int \vee bool \mid true)$ does not hold. However, the subtyping relation $int \leq int \vee bool$ holds.

Since subtyping has to be defined on infinite terms, we adopt a coinductive definition [11,12]. We define $\leq$ as the greatest binary relation defined on the restricted co-Herbrand universe satisfying the following rules:

$$(\text{bool})\frac{}{bool \leq bool} \qquad (\text{obj})\frac{\overline{t_1 \leq t_2}^{\,m}}{obj(c, [\overline{f{:}t_1}^{\,n}]) \leq obj(c, [\overline{f{:}t_2}^{\,m}])}\ n \geq m$$

$$(\vee\text{R1})\frac{t \leq t_1}{t \leq t_1 \vee t_2} \qquad (\vee\text{R2})\frac{t \leq t_2}{t \leq t_1 \vee t_2} \qquad (\vee\text{L})\frac{t_1 \leq t \quad t_2 \leq t}{t_1 \vee t_2 \leq t}$$

$$(\text{distr})\frac{obj(c, [\overline{f{:}t}^{\,n}, f{:}t_f, \overline{f'{:}t'}^{\,m}]) \leq t \quad obj(c, [\overline{f{:}t}^{\,n}, f{:}t'_f, \overline{f'{:}t'}^{\,m}]) \leq t}{obj(c, [\overline{f{:}t}^{\,n}, f{:}t_f \vee t'_f, \overline{f'{:}t'}^{\,m}]) \leq t}$$

Lemma 3. *If* $cds \rightsquigarrow P$, e ***in*** $V \rightsquigarrow (t \mid B)$, $B\theta \subseteq M^{co}(P)$, *and* $e \rightarrow_{cds} e'$, *then there exist* t', B' *and* θ' *s.t.* e' ***in*** $V \rightsquigarrow (t' \mid B')$, $B'\theta' \subseteq M^{co}(P)$, *and* $t'\theta' \leq t\theta$.

Theorem 2 [Subject reduction]. If cds $e \rightsquigarrow (P, B)$, B is satisfiable in P, and $e \rightarrow_{cds} e'$, then cds $e' \rightsquigarrow (P, B')$, and B' is satisfiable in P.

Proof. A corollary of lemma 3. □