Private types in Higher Order Logic Programming

Private types in Higher Order Logic Programming

Marco Maggesi
Università degli Studi di Firenze, Italy
Marco.Maggesi@unifi.it

Enrico Tassi
Inria, Université Côte d'Azur, France
Enrico.Tassi@inria.fr

We report on ongoing work on introducing a mechanism for private types in a higher-order logic programming language such as λProlog.

1 Private types in OCaml and their application

Algebraic data types are pervasive in both logic programming (LP) and functional programming, even more, they are one of the characterizing features of these families of languages. When developing large or critical software, the programmer often needs to hide the actual definition of a type, in order to gain modularity or ensure invariants by construction. Both families of languages provide ways to seal type declarations in a module signature. We are mostly interested in enforcing invariants in the context of symbolic computation. A typical example of code we are interested in is the kernel (trusted component) of a prover, the use case that motivated sealed types in ML in the first place.

There is a tension between ease of use of algebraic types, e.g., language support for matching or unification, and sealed types that can only be built via an API of constructors or inspected via an API of views. OCaml provides an elegant compromise: *private types* [1, Section 8.3]. We illustrate their use in the following code snippet taken from the kernel of HOL Light.

```
module Hol : sig

  type term = private
    | Comb of term * term
    | ...

  val mk_comb : term * term -> term
  val dest_comb : term -> term * term

end = struct

  let mk_comb (f,a) =
    match type_of f with
    | Tyapp("fun",[ty;_]) when compare ty (type_of a) = 0 -> Comb(f,a)
    | _ -> failwith "mk_comb: types do not agree"

  let dest_comb t =
    match t with
    | Comb(f,a) -> f, a
    | _ -> failwith "dest_comb: not an application"

end
```

The code defines the datatype of terms that are well-typed by construction. The `mk_comb` function is the only constructor for application (named combination in HOL) and enforces that the type of the

Submitted to:
TASE-LP

argument matches the type expected by the function. In order to prevent the client from building an application without calling `mk_comb`, one could seal the type `term` in the signature by hiding its constructors. However, one would also be forced to define APIs such as `dest_comb` to let the client actually do something useful with terms. In OCaml, making the type *private* is sufficient to enforce that expressions of type `term` can only be built by using the API in the module signature but does not prevent the client from inspecting terms using the `match-with` linguistic construct. As a consequence, the API `dest_comb` at line 8, is superfluous, strictly speaking. We now give two simple examples of OCaml code featuring allowed and forbidden language constructs outside the defining module `Hol`.

```
1  (* accepted: pattern matching is allowed on private constructors *)
2  let operand t =
3    match t with
4    | Hol.Comb(f,x) -> x
5
6  (* rejected: Hol.Comb is a private constructor *)
7  let apply_twice f x = Hol.Comb(f, Hol.Comb(f, x))
8
9  (* accepted: uses the safe constructor "mk_comb" that checks at
10              run time the well-formedness of "f (f x)"           *)
11 let apply_twice f x = mk_comb(f, mk_comb(f, x))
```

We stress that the mechanism of *private* types is different from the one of *opaque* types which forbids the use of constructors for both pattern matching and term building outside the defining module. To the best of our knowledge only opaque types have been considered in the context of logic programming.

2 Private types in LP based on modes and type checking

The implementation of private types in functional programming languages reposes essentially on the fact that the operations of constructing terms and matching (or destructing) terms are syntactically distinct. The occurrence of `Comb` at line 14 constructs a term while the occurrence at line 19 inspects a term. When a type is private, the former operation is only allowed in the module in which the type is defined, `Hol` in the example. Instead, destruction of terms is allowed everywhere.

In contrast, in LP languages, the action of inspecting data and building data are conflated into unification, which is a cornerstone of the relational semantics of logic programming. Still, many LP systems let the user annotate predicates with their intended modes. As shown in [2, Section 3], in then the execution of a well-moded program unification is equivalent to matching if this condition holds: *(*) the query arguments in input position are ground*. This observation suggests a way to introduce private types in LP based on the well-understood mechanisms of modality and typing.

Definition 1 (private type) *A type T is private in a module M, if all clauses $p\ A_1 \ldots A_n \leftarrow H$ in M are such that constructors of T: (1) do not occur in the premise H and (2) if they occur in argument A_i to the predicate p in the head of the clause, then the i-th argument of p is declared to be an input.*

The notion of private type given by conditions (1), (2) and (*), is purely syntactical and can be enforced at type-checking. In particular, this notion enjoys two fundamental properties: (i) it does not introduce a computational cost at runtime; (ii) it does not change the declarative semantics of programs.

3 Work in progress: Higher Order features

The syntactic criteria 1) and 2) do not trivially transfer to the higher order case, as shown by this example:

```
1  type comb tm -> tm -> tm.      % the private constructor
2  type mk-comb tm -> tm -> tm.   % the safe constructor
3
4  mk-comb F A (comb F A) :- type-of F (tyapp "fun" [S,_]), type-of A S.
5
6  evil T :-
7    pi f\ type-of f (tyapp "fun" [nat,nat]) =>
8    pi a\ type-of a nat =>
9      mk-comb f a (Leak f a),
10     T = Leak z z.
```

In the code of the predicate `evil` there is no occurrence of the private constructor `comb`, but it still manages to synthesize an ill-typed term `T` (the application of `z`, a term of type `nat`, to itself). Higher-order unification at line 9 extracts the head symbol out of a well-typed term built using the safe constructor `mk-comb`. Indeed the variable `Leak` gets assigned the value `comb` at run time. The question we hope to answer to is: can we find a static criteria to implement private types in higher-order logic programming?

References

[1] *The OCaml system release 4.10. Documentation and user's manual*. https://caml.inria.fr/pub/docs/manual-ocaml/privatetypes.html. Accessed: 2020-03-02.

[2] K.R. Apt & E. Marchiori (1994): *Reasoning about Prolog programs: From modes through types to assertions*. *Formal Aspects of Computing* 6, pp. 743–765, doi:https://doi.org/10.1007/BF01213601. Available at https://link.springer.com/article/10.1007/BF01213601.