Type Inference Using Meta-extract for Smtlink and Beyond

Yan Peng Mark R. Greenstreet
University of British Columbia
Department of Computer Science
Vancouver, Canada
yanpeng,mrg@cs.ubc.ca

We present clause processors for type inference of ACL2 terms. The motivating application is Smtlink: we need a bridge from the untyped logic of ACL2 to the many-sorted logic of SMT solvers. Unlike the ACL2 `type-alist`, our clause processors support user-defined types. Furthermore, we introduce a `typed-term` data structure that associates type-judgements with subterms of a clause. This means that when syntactically identical subterms can occur in logically distinct contexts, our clause processors can make distinct type-judgements for each occurrence of the subterm. These capabilities make these clause processors useful for applications beyond Smtlink.

The logic of ACL2 is untyped, whereas the logic of SMT solvers is many-sorted. The soundness argument of `Smtlink`[3] relies on a notion of "type-hypotheses": there should be a hypothesis asserting the type of each free-variable appearing in a theorem; a verified clause processor shows that the theorem trivially holds when variables have values outside their intended domain; and the remaining "typed goal" is then discharged by the SMT solver. Our new clause processors improve on this type deduction in three ways. First, missing type-information is detected early in the translation pipeline, allowing for clearer error reporting. Second, `Smtlink` the support for user-defined types is now verified – using FTY is convenient, but no longer required or trusted. Third, users were required to annotate their goals with type information by using fixing functions; now most of these type judgements are inferred automatically.

ACL2 has `type-alist`[1] which records type information. While, the `type-alist` checks for a small, fixed set of properties used by the ACL2 proof engine, our clause processors support a wide range of user-defined types. The `type-alist` is an alist that maps pseudo-terms to type-sets. If the same pseudo-term occurs in multiple contexts, different type-judgements may occur for each instance. We introduce a new data structure, `typed-term`, that records context-dependent type judgements.

The Type Inference Algorithm and the Clause-Processors

We define a new data structure `typed-term`.

```
(defprod typed-term ((term pseudo-termp) (path-cond pseudo-termp) (judgements pseudo-termp)))
```

The `term` field represents a clause or a subterm of the clause. `path-cond` is the conjunction of branch conditions that leads to `term`, and `judgements` are the type-judgements for `term`. For example if a goal includes the hypothesis `(integerp x)` and has the subterm `(* x x)`, the typed-term for `(* x x)` is

```
(typed-term '(binary-* x x) '(integerp x) '(if (if (integerp (binary-* x x)) 't 'nil)
                                               (if (if (integerp x) 't 'nil)
                                                   (if (if (integerp x) 't 'nil) 't 'nil)
                                                 'nil)
                                             'nil))
```

The shape of the judgement maps directly to the shape of the term.[1] The judgement is added as a hypothesis to the main goal. We provide a set of constructor and destructors for `typed-terms` that make

[1] Note that for each occurrence of `x` in `(binary-* x x)`, there is a type judgment for it.

G. Passmore, R. Gamboa (Eds.): ACL2 Workshop 2020
EPTCS ??, 2020, pp. 1–3, doi:10.4204/EPTCS.??.??

it easy to access subterms, their path conditions, and their type judgements.

The clause processors are invoked with an argument that lists the type-signatures for functions (by listing the relevant "returns" theorems), and the type-recognizers to be used for type-judgements, along with supertype/subtype relationships (again, specified by naming the relevant theorems). For use in Smtlink, type-inference consists of three clause processors. The first clause processor performs a post-order traversal of the goal, and constructs a `typed-term` that annotates each subterm with *all* type recognizers (from the provided list) that it can be shown to be satisfied by the subterm. Supertype and subtype judgements are added taking into account the path condition. The clause processor recursively traverses the pseudo-term for with the following five cases:

- Constants: Types of constants are trivially proved using meta-extract[2].
- Variables: We find type judgements from the path condition. The disjuncts of a clause are sorted so that "type hypotheses" such as `(integerp i)` appear early in the path. A "type-judgement" is an expression with only one free variable that is implied by the path condition.
- `if`-expressions: We treat `if` specially because the condition of an `if` is used to augment the path-condition for the "then" and "else" arguments.
- Lambdas: We properly shadow the path condition and pass type judgements of the actuals into the formals in the body to infer types in the body.
- Function calls (other than `if`): Return type information is obtained using meta-extract. Polymorphism is supported by allowing the same function to have multiple `:returns` theorems.

The second and third clause processors are SMT specific. Having found *all* valid type-judgements for each subterm with the first processor, the second clause processor chooses type-judgements that are consistent for the many-sorted logic of the SMT solver. For example, `(list 1 2 3)` satisfies both `integer-listp` and `rational-listp`, but Z3 considers these to be distinct sorts. Note that this clause processor "chooses" by discarding the type-judgements (i.e. disjuncts of the clause) that won't be used for the translation. A third clause processor is introduced to do static dispatch of polymorphic functions. For example for function `car`, we define its typed version `integer-list-car`:

```
(defun integer-list-car (x) (if (consp x) (car x) (ifix x)))
```

The clause processor uses meta-extract with a "replace theorem" as shown below to justify replacing a call to `(car term)` with a call to `(integer-list-car term)`.

```
(defthm replace-of-car
  (implies (and (integerp-list-p x) (consp x)) (equal (car x) (integer-list-car x))))
```

By combining type judgements inferred for `term` and the path condition establishes the hypotheses for the theorem automatically (in most cases). As a fall-back, the user can add fixing functions to force type assertions. The clause processor can generate a new term that is well-typed for Z3 and provably equivalent to the original term.

Conclusion and Future Work

We implemented three clause-processors for type inference, type unification, and term transformation. These clause-processors will allow `Smtlink` to infer type information for translation of goals to the many-sorted logic of an SMT solver. We are currently completing the correctness proofs of these clause-processors and will apply them to real examples.

References

[1] Robert S. Boyer & J. Strother Moore (1979): *A computational logic handbook*. *Perspectives in computing* 23, Academic Press. Available at http://www.cs.utexas.edu/users/boyer/acl.pdf.

[2] Matt Kaufmann & Sol Swords (2017): *Meta-extract: Using Existing Facts in Meta-reasoning*. In Anna Slobodová & Warren A. Hunt Jr., editors: *Proceedings 14th International Workshop on the ACL2 Theorem Prover and its Applications, Austin, Texas, USA, May 22-23, 2017*, *EPTCS* 249, pp. 47–60, doi:10.4204/EPTCS.249.4.

[3] Yan Peng & Mark R. Greenstreet (2018): *Smtlink 2.0*. In Shilpi Goel & Matt Kaufmann, editors: *Proceedings of the 15th International Workshop on the ACL2 Theorem Prover and Its Applications, Austin, Texas, USA, November 5-6, 2018*, *EPTCS* 280, pp. 143–160, doi:10.4204/EPTCS.280.11.