Abstract

This paper summarizes an attempt to add a typing mechanism to ACL2. Types are only used to reject ill-typed functions, but it is argued that it would not be hard to modify the system to use type information for theorems as well. This requires modifying the ACL2 system code; the author argues that ACL2 could be modified to allow such experimentation without putting the soundness of the system in question. The type checker described contributes nothing new to the extensive literature on type systems for functional languages.

Adding a typing mechanism to ACL2

Vernon Austel
IBM T.J. Watson Research Center

November 16, 2004

1 Introduction

This article describes a relatively simple type system that the author has found useful, in that it assigns a useful type to most functions that the author writes and has caught errors. It is definitely not claimed to be a *good* type inference system, or that it should be adopted by the ACL2 community, or even that it works as advertised; it was developed for the author's own use and is offered simply as a proof of concept. Examples of sophisticated type systems for lisp include Soft Scheme[8], the system by Akers[1] and that of Widera and Beierle[7]; less sophisticated are STYLE[6], the system of Jenkins[4] and that of Kind and Friedrich[5]. Much more work on type systems has been done in the context of ML.

The real contribution of this article is not the simple type system, but rather the insights the author gained during its integration into ACL2 that prompt him to make the following (unsuppported) claims.

First, the maintainers of ACL2 could write a framework that allowed others to experiment with type systems without having to hack the system code.

Second, this framework could allow type information to be used by the ACL2 proof system in a way that raises soundness issues only concerning the processing of embedded events.

Third, creating this framework would not require a great deal of work from the maintainers of ACL2.

Finally, it is claimed that the system should be able to work in both a typed and untyped mode; the addition of a type system need not inconvenience those who prefer an untyped mode. No existing ACL2 books need be changed to accomodate one.

The paper concerns three related topics. Sections two through 4 outline the primitive type system the author has developed, just to give a flavor of what one might look like. Section 5 gives a brief overview of how types would be useful in ACL2. Section 6 proposes a general approach to adding type systems to ACL2, and sections 7 through 8 proposes a general extension to the ACL2 macro facility to support that approach. Knowledge of the author's type system is not required for the rest of the paper.

2 The type language

In this system, a type is:

- a symbol, or
- a quoted constant (that must be an atom), or
- a non-negative integer, or
- the list `(pairp` tp1 tp2`)`, where tp1 and tp2 are types, or
- the list `(listp` tp`)`, where tp is a type, or
- the list `(or .` tps`)`, where tps is a true list of types

For example, the literal `('x 'y)` is a member of the type `(listp atom)`, but also of the type `(listp symbolp)`, and `(or integerp characterp (listp (or 'x 'y)))`.

A type that is a symbol is called a type name. Every type name corresponds to an ACL2 predicate that tests for membership in the type. Generally, these predicates have a single argument, but in some cases they may have two (discussed below). ACL2 functions may be made type names using the new event `define-type`. There are built-in type names corresponding to many ACL2 type-prescription types (`integerp`, `acl2-numberp`, etc), but there is nothing special about them; they are introduced to the system using `define-type` (see below).

A quoted constant denotes the type such that only that constant is a member. This is just a convenience; clearly any such type may be defined by a type name.

Natural numbers are used to represent type variables, simply because it seemed convenient for the implementation.

The lisp `cons` functor provides the functionality of both a pair type and a list type, and sophisticated type systems can determine from context how `cons` is being used, but the system described here lacks such sophistication. Instead, it provides the functions `pairp`, `pair`, `fst` and `snd` that correspond to `consp`, `cons`, `car` and `cdr`, but concern pairs rather than lists.[1] Some would argue that this style is clearer anyway. The list-type constructor is called `listp`, even though that function already exists in Common Lisp and may seem inappropriate for use as a type constructor for true lists. There is no constructor for improper lists.

The type of a function is represented by a non-empty true list of types; the first type is the return type, and the rest are the argument types. Again, this was chosen for implementation convenience.

[1]This also required changing the implementation of multiple values slightly, since they are now nested pairs instead of a true (untyped) list.

2.1 Subtypes

Some simple subtype information is built-in to the system; for example, `'nil` is known to be a subtype of `booleanp`, and `tp` is known to be a subtype of `(or tp tp2)` for any type `tp`.

The user may introduce new subtype relationships among type names using the new rule class `:SUBTYPE`. The implemention of subtype relationships is modeled after that of refinement relationships.[2] A new macro called `defsubtype` call is analogous to the macro `defrefinment` call. Obviously, only type names may be used in subtype theorems.

2.2 Defining new type names

The user may define arbitrary types using `define-type`. For example, a predicate recognizing valid C identifiers (where identifiers are represented by ACL2 symbols in package ACL2) could be made a type:

```
(defun c-symbol-p (x)
   ;; test if (symbol-name x) is a valid C identifier
   <elided for brevity>)
(define-type c-symbol-p)
(defsubtype c-symbol-p symbolp)
```

In such cases, there are no constructors for the type; instead, one uses the predicate to separate members of the type from non-members. The return type of this function will be `(listp c-symbol-p)`:

```
;; takes a symbol list and returns the c-symbol-p sublist
(defun filter-c-symbol-p (l)
   (cond ((endp l) nil)
         ((c-symbol-p (car l)) (cons (car l) (filter-c-symbol-p (cdr l))))
         (t (filter-c-symbol-p (cdr l)))))
```

2.3 state-p

The ACL2 state, which naturally has type `state-p`, was made abstract. That is, state-accessing and -modifying functions in axioms.lisp are assigned types that involve the type `state-p` rather than the type that would ordinarily have been assigned, which would be some kind of nested pair. The ACL2 expression reader ensures that the state is properly handled in function definitions, so this in no way contributes to type-correctness, but it does make the types inferred for these functions more concise. Otherwise, no special support for state has been necessary.

[2]In fact, the code maintaining the SUBTYPES property is clearly derived from the code implementing equivalences.

2.4 Setting a function's type

Ordinarily, the type system infers a function's type from its body; the user may optionally provide a type using a new `xargs` keyword, which is checked to see if it works. There are two occasions where this won't work, in which case one must use the new event `set-function-type` to assign a type to the function. The first case is where the function was not assigned a type when it was defined; this can happen either because type-checking mode was turned off at the time, or because an `xargs` keyword was used to prevent a type from being assigned. If `set-function-type` is passed nil (which is not a valid function type), it attempts to infer the type of the function from its body. This is useful for bootstrapping; most of the functions defined in axioms.lisp are assigned types this way. Otherwise, `set-function-type` must be passed a valid type, and makes it the function's type without any check or proof. Such a proof would require translating the syntactic type to an ACL2 proposition corresponding to it; some work along those lines was done, but the author didn't complete it.

2.5 anytypep

A special type `anytypep` plays the role of the top element in the type system; every ACL2 object has this type.[3] It is necessary for some functions that return objects from the ACL2 state, for example, `get-global`:

```
   (set-function-type get-global (anytypep symbolp state-p))
```

It is also necessary for functions that take a list of values of arbitrary type as input, in particular, format lists. An example of this is `hard-error`:

```
(set-function-type hard-error
     ('nil symbolp stringp (listp (pairp characterp anytypep))))
```

The last argument type represents the standard ACL2 alist that maps characters to arbitrary values. One might consider using a type variable rather than `anytypep`, but that would force all objects in the range of the alist to have the same type, which is clearly not what we want.[4]

When a function type is derived, type variables that are used only once are replaced by `anytypep`. The effect is the same, but hopefully makes the type clearer (by highlighting the type variables that really are used) and probably speeds up the type inference mechanism a little.

2.6 Multiple values

Multiple values are different only in that they must be nested pairs rather than fixed-length lists, since the values returned must not be required to have the

[3]Following the implementation of equivalences, every type name in the system appears in the list of subtypes of `anytypep`.

[4]It isn't clear if it is a good idea to allow user-defined types to be printed that way. If not, then instead of `anytypep`, these format alists should a type that represents everything but non-atomic user-defined types.

same type.[5] The return type for such functions is cumbersome, since the current system does not have a version of `pairp` that takes arbitrary number of arguments; this wasn't considered important enough to implement at the time. Some special support seems to be required for multiple return values in the type inference algorithm, but not much; they are generally just treated as (nested) pairs.

This is an example of a function with multiple return values:

```
(set-function-type read-char$
    ((pairp (or 'nil characterp) (pairp state-p 'nil))
     symbolp
     state-p))
```

Note that the final pair must have `'nil` as its second element.

3 Defining recursive types

Sophisticated type systems can deduce recursive types directly from function definitions, but this simple system cannot; recursive types must be introduced by the user using the macro `defntype`.[6] This macro is not a primitive event, but rather expands into code that calls `define-type` and `set-function-type`.

The following example of a user-defined type models C types:

```
(defntype ctype
  (basic-type symbolp)
  (ptr-type   basetype ctypep)
  (ref-type   basetype ctypep)
  (array-type basetype ctypep
              nelems   integerp)
  (struct-tp ids (listp symbolp)
             decls (listp ctypep)))
```

The syntax is what one would expect from such a utility. The new type name is `ctypep`; each item after the name defines a new type variant. Each variant has a symbol that serves as a name (e.g. `basic-type`), after which should follow any number of name/type pairs for the variant's components. Each variant fields has an accessor that is the concatenation of the variant name and the field name, so for example `ptr-type-basetype` accesses the (only) field of the `ptr-type` variant.

In addition to simple recursion (as in `ptr-type`), this system also allows recursion on lists of the new type (as in the `decls` field of the `struct-tp` variant). No other kind of recursion is allowed. Obviously, this is a very severe restriction, but it simplified the implementation and has been sufficient for the author's purposes.

[5]This required changing some of the system code to use the pair functions `pair`, `fst` and `snd`.

[6]This macro can also introduce non-recursive record types, of course.

A function `ctype-kind` is generated that returns one of the variant names (e.g. `basic-type`) when it is applied to an object of type `ctypep`.[7] This function is used to distinguish between variants in functions defined on the new type. This speeds up rewriting, but slows down the execution of the function.

Functions that operate on types that use list recursion (such as `ctype` above) have more complex recusive definitions. There are two standard ways to deal with this in ACL2. Either one may define two functions using `mutual-recursion`, or one may define a single function that uses a flag to distinguish between the cases; see the `mutual-recursion-proof-example` topic in the ACL2 documentation. The author choose to use a single function with a flag to implement these cases. In the author's experience, while it is possible to use mutually recursive functions in proofs, it is cumbersome, and since the goal is to use these functions in proofs rather that to write efficient system code, we prefer the style that is easier for proofs.

To support this, type predicates are allowed to have two arguments, but they must conform to this pattern:

```
(defun some-predicate (not-listp x)
   (if not-listp
       <some expression in which ‘‘not-listp’’ is not free>
     <a list-recursive expression in which ‘‘not-listp’’ is not free>))
```

Functions defined on such recursive types must also conform to this pattern. Recusive calls must be passed `t` or `nil` rather than the first argument (here: `not-listp`). Such functions cannot be typed in the usual way; if the system cannot type a function the usual way, it checks whether the function has this pattern and if it does then it types it using a different, more complicated way to accomodate list recursion.

3.1 Defining a function on the new type

An example of a function defined on `ctypep` is given below. When `defntype` is used to define a new type, it defines a convenience macro for defining functions on the new type; in this case, the new macro is `defctype`. The macro is not part of the type system, it just eliminates some of the tedium of writing such functions. Since `ctypep` is list-recursive, the first argument must be the flag discussed above. The list cases for such functions are given by `:NULL` and `:CONS`; these names are predefined and should not be used for other purposes. The `defctype` macro allows the user to refer to subcomponents of the argument (`ctype`) using predefined names (e.g. `basetype`); this is the same style used by the author in a previous attempt to implement types in ACL2.[2].

```
(defctype print-type (ctypep ctype channel state)
  :BASIC-TYPE   (princ$ ctype channel state)
  :PTR-TYPE     (pprogn
```

[7]Such functions return the symbol `MALFORMED` when applied to objects of the wrong type. In principle, that should never happen.

```
                (print-type t basetype channel state)
                (princ$ #\Space channel state)
                (princ$ "*" channel state))
;; other cases elided for brevity
:NULL          state
:CONS          (pprogn
                (print-type t car-ctype channel state)
                (print-type nil cdr-ctype channel state)))
```

4 Difficulties

This section points out features that the author did not address with this simple system.

4.1 Arithmetic operators

ACL2 has only one set of arithmetic operators. A sophisticated type system would be able to model the fact that the `+` function returns an acl2-number in general, but returns an integer if both its arguments are integer; this simple type system does not. Instead, it just uses two sets of operators: the usual ones (`+`, `-`, etc) for integers and a second set (`f+`, `f-`, etc) for rationals.

4.2 Natural numbers

The author briefly experimented with a type for natural numbers; this would allow one to define the type of `nth` to be `(0 naturalp (listp 0))`. It quickly began to seem more trouble that it is worth. It has the same problems with arithmetic operators mentioned above, and it seems to require the system to use linear arithmetic. The user may always define such a type using `define-type`, of course, but it may not be as useful as expected, since there is no special support for it.

4.3 ACL2 arrays

ACL2 “arrays” pose a challenge for the type system. Roughly speaking, ACL2 arrays are a logical interface to Common Lisp arrays. The logical representation is almost an alist mapping natural numbers to values, except that one element in the list (the so-called header) that contains information needed to keep the logical representation synchronized with the underlying CL array implementation. Again, sophisticated type systems could give them a useful type, but this system cannot. For the time being, the author just does not use arrays.

4.4 Stobjs

The author currently does not use stobjs in his work. Since this type system was built to meet the author’s current needs and is not an end in itself, he has made no effort to deal with them. It seems likely that they would be treated much as the ACL2 state is: a new primitive type would be defined, the stobj accessors and updaters would be assigned the appropriate types, and things would work as usual from there. However, there is no experience to back up this claim.

5 How types can be used

There are three ways a type system would be useful in ACL2: as a simple check for function definitions (just like in typed programming languages); to generate guards; and to allow the statement of theorems to be more concise. A type system need not do all of these things, of course.

5.1 Types for sanity checking

At the very least, a type system must assign a type to new functions, or reject them as ill-typed. A type system that just does this is still useful, because it gives some assurance that functions do reasonable things.[8] Since the only way such a system can influence ACL2 is by rejecting function definitions, it is easy to see that it cannot impair the soundness of the system, even in embedded events.[9]

Most type systems either assign a type to a function or reject it as ill-typed. In contrast, a *soft-typing* system nevers consider a function ill-typed; it assigns a type to all functions, but it may also modify the function submitted by the user so that it generates a (fatal) error under some conditions.[3] The motivation for this is to avoid executing machine operations that cause errors or undefined behavior; since ACL2 by definition has no such operations, this would not seem to apply, but nevertheless this might be worth investigating. This would have soundness issues concerning embedded events, since it modifies the function rather than just assigning it a type, but the issue is similar to the one concerning theorems and would have similar solutions.

5.2 Types for guards

For many functions, the function type generated by a type system can be used to generate its guard; in such cases, it seems reasonable for the type system to add this guard to the function definition. For example, the guards of most list-manipulation utilities could probably be generated by a type system, whereas anything that used `nth` probably could not.[10] The author's impression has

[8]This is all that the author's type system does.

[9]At the worst, it might accept a function in one pass and reject it in the other; that would be annoying, but does not affect soundness, since the event will just fail.

[10]In contrast, a type system *could* use a dynamically checking version of `nth`, that is, one that causes an error if its integer argument is not less than the length of its list argument, and whose guard does not require that. This is what typed languages such as ML and Java do. This would allow those who don't need peak performance to get good performance easily (by having guards automatically generated and verified).

always been that the ACL2 guard mechanism is cumbersome; any means of simplifying it should be welcome.[11]

Note that the type system would only generate the guard, it would not replace the guard proof; a good type system should ensure that the guard proof for guards it generates always succeeds, by generating appropriate lemmas and ensuring that they are enabled.

For any type system there will probably be functions such that the function type cannot serve as a guard. Nevertheless, it may be helpful to use the type system for this purpose the rest of the time.

A type system that generates guards could in principle raise soundness issues if it is used in embedded event forms.[12] The issue is similar to the one for theorems and would have similar solutions.

5.3 Types for concise theorems

For someone accustomed to typing, ACL2 theorems seem unnaturally verbose; all the type information must be explicitly stated in the theorem rather than being implied by the functions used. Consider the following:

```
;; this event fails in ACL2
(defthm append-nil
  (equal (append x nil) x))
```

This event fails — the goal is not a theorem in ACL2.[13]

However, it isn't hard to imagine a macro such that a form with very similar looking text would succeed:

```
(typed-defthm typed-append-nil
  (equal (append x nil) x))
```

Such a macro would examine the goal and add type hypotheses as needed, so for example the above form might expand into the following:

```
(defthm typed-append-nil
  (implies (true-listp x)
           (equal (append x nil) x)))
```

Although the event the user types has no hypotheses, the goal proved does, as does the rule derived from it.

In addition to making theorems more concise, one could imagine a type system modifying the prover trace output by dropping type-related literals. For example, in the above theorem `typed-append-nil` there is no point in seeing

[11] It may even be possible to have an untyped version of ACL2 that uses the type system solely for guard generation; this might even be popular with those who see no value in type checking.

[12] It may seem that guards are unrelated to soundness, but it isn't; see the discussion of safe mode in interface-raw.lisp.

[13] This can be quite confusing for new users of ACL2, since they assume that this proposition must naturally only concern lists.

the hypothesis `(true-listp x)`, since that is generated by the type system, so it might be useful for the type system to remove it from the printed goal. This concerns only what is printed, of course; the literal would still be in the goal used during the proof.

5.4 Typing cannot affect soundness

Assume that the following form succeeds:

```
(typed-defthm does-this-show-that-ACL2-is-unsound?
  (equal (append x x) x))
```

Could this be used as evidence that ACL2 is unsound? After all, this is not true, even if `x` is a true list.

Of course not. This is not evidence that ACL2 is unsound; it is merely evidence that this alleged typing macro is not useful. The macro may have generated `nil` as a type hypothesis; such a rule will never actually be used.

Can theorems proved using such a typing macro interoperate with untyped theorems? Of course they can; the question doesn't really even make sense, since there is no difference between the two internally. The only difference is in the text that the user types.

Should any ALC2 user be concerned about how a typing macro works? Should such macros be investigated to determine whether or not they impair the soundness of ACL2? Of course not; by design, the ACL2 macro facility is such that no macro can have such an affect. A macro may prove to be misleading or not useful, but it cannot affect the soundness of ACL2.

Can an ACL2 macro function as a general-purpose typing utility? Unfortunately not.[14] A usable type system must constantly extend the set of functions whose type it knows about; this seems to require storing type information in the ACL2 world, which macros currently cannot do. However, in spirit it is no different from a macro; it performs no proof, it just generates text.[15]

Unfortunately, in the current design of ACL2 any macro or function that accesses the ACL2 world can potentially cause unsoundness; this general design characteristic is the only potential source of unsoundness for the kind of typing framework proposed here. The framework proposed here can use untrusted code to implement the type system so long as the design of ACL2 is changed to avoid this problem; we propose a solution in a later section, but other solutions may be possible.

[14]In particular, a macro cannot perform macro expansion on a function body, which may be necessary in order to type it.

[15]There is one situation where it might be expected to do more. Some type theorems are really polymorphic; for example, append takes two lists and returns a list. The type of each element in the returned list is either the element type of the first list or the element type of the second. This cannot be expressed in a single theorem in ACL2. A theorem may require a type theorem concerning append for specific argument types. The simplest thing to do is require the user to somehow indicate to the type system that they must be generated; however, in principle the prover could be modified to detect when a proof attempt fails due to a missing type hypothesis, in which case it would generate that hypothesis, prove it, and retry the main proof.

6 A framework for experimenting with types

We believe that a great deal of experimentation may be required to find a type system appropriate for ACL2, and that the best way to foster such experimentation is to allow the type system to be untrusted. We believe that this is not hard to do and raises no soundness concerns, with the exception of the issue of embedded events. We propose one technique to deal with that issue, but if this technique is somehow flawed or is considered undesirable, perhaps some other solution to the problem can be found.

We propose allowing macros to have access to the ACL2 world, and adding an event to update the ACL2 world. Although the type checker described in this paper was not implemented this way, we claim (without proof) that if both these features were added to ACL2, then something very similar to this type checker could have been implemented without hacking the system.[16] The argument isn't that this is the best way to implement type checking in ACL2; the argument is that this extension not very hard and is enough to enable experimentation of all kinds, including type checking.[17] The only soundness issues would concern the two extensions to ACL2, not the type system.

7 How to permit macros to access the ACL2 world

ACL2 only allows certain forms inside `encapulate` forms or inside a book; such events are called *embedded events*. The restrictions are required because these events are processed twice, the first time to prove the events, the second to modify the current world (whereby local events are ignored); in the case of `encapsulate`, the second pass immediately follows the first, but in the case of `include-book` the first pass is the certification phase, while the second pass occurs when the book is loaded.[18]

The documentation topic `embedded-event-form` gives an example of how ACL2 could be rendered unsound if one did not make some kind of restriction.

```
(if (ld-skip-proofsp state)
    (defun foo () 2)
    (defun foo () 1))
```

ACL2 system code "knows" which pass is being processed using `(ld-skip-proofsp state)`; in the first pass, it will be `nil`, but in the second pass it will be

[16]There would be minor differences. The type checker described here actually changes the code for `defun`, whereas the proposed style would have to use a macro such as `defun-typed`. Also, there would have been no new :SUBTYPE rule class, since that involves setting a system property. That is unimportant; the information could have been stored differently.

[17]Of course, people can experiment with such things at the top-level, but that is very unappealing. No one is going to put much effort into a feature that cannot be used in books, since everything eventually ends up in a book.

[18]The system also does a second pass during certification.

`'include-book`. Users could exploit this difference in `state` to cause unsoundness; the system would prove one set of events in the first pass and then assume the correctness of different events in the second. Although this example concerns `state`, the same argument holds for the ACL2 world.

The ACL2 documentation for `defmacro` implies that macros are not allowed to have access to `state` to keep in the spirit of CLTL; this makes their expansion independent of the current environment. However, it seems clear that the real problem is that issues similar to those for functions can arise; that is, a macro accessing `state` (or the world) could expand one way during the first pass and another way during the second. Unlike the case above, though, there might be a way to prevent this from causing unsoundness.

The central idea is to save something from the first pass to confirm that the second pass expanded the same way; so long as it does, there can be no soundness issues. There are different ways to do this, each with its own problems.[19]

The obvious way is to save the result of macro expansion in the first pass and re-evaluate it in the second. This guarantees that the user-defined macros won't return a different expansion, since they are only evaluated once. The problem is that the result of macro expansion cannot be an embedded event (since embedded events are macros); for example, `defthm` expands into a call to `defthm-fn`. It seems reasonable to just drop the syntactic checks for the second pass; after all, the form passed the checks in the first pass.

If that technique is rejected, perhaps the first pass could save only a partially-expanded form for re-evaluation. A form is safe so long as it does not contain user-defined macros that take the world or state as a parameter. In the first pass, a special kind of translation function could macro-expand forms only until this point is reached; this partial expansion would be saved, and then it would be fully expanded for the first pass to evaluate. The second pass would read the partially-expanded form and also fully expand it. The advantage would be that the saved form would still conform the existing syntactic requirements; the disadvantage is that it would require a special kind of macro expansion function.

Finally, the first pass might save the fully expanded form, but the second pass would not directly evaluate it. Instead, it would have both the form from the book and the expansion saved from the first pass; it would use the saved expansion only to confirm that the form expanded to the same thing. This would require awkward changes in the code that deals with the second pass; it is unlikely that this would be preferable to the first technique.

8 An event that can change the world

It seems safe to allow write access to the world as well as read access. We propose a new event corresponding to `putprop`:

[19] All of these have the problem that certification would produce a file that `include-book` critically relies on. However, this should not be conceptually different from trusting the validity of the certificate. Also, the `local` event macro doesn't expand to the same thing in the two passes; it would have to be changed. There may be other such macros.

```
(putprop-event <symb> <key> <value>)
```

This event would naturally ensure that the new world is an extension of the old one and is well-formed. In addition, it would reject as an error any attempt to modify system properties (such as `'type-prescription`); this could perhaps be accomplished by making this part of the macro's guard.

If such a general-purpose event is not added, then other special purpose events will probably be needed by a type system, for example to introduce new type names and subtype relationships.

9 Conclusion

This paper presents a simple type system that type-checks ACL2 functions. It argues that the addition of a type system would not require modifications to existing libraries and would not affect those who want to used ACL2 in an untyped fashion. It claims that a useful type system could be implemented using macros if ACL2 is modified so that macros are allowed to access the world and an event is added that allows the world to be updated in a restricted fashion. Finally, we claim (without proof) that it would not be difficult for the maintainers of ACL2 to make these changes, and that they would foster experimentation of all kinds, not just type systems.

References

[1] Robert L. Akers. *Strong static type checking for functional common lisp.* PhD thesis, University of Texas, Austin, 1993.

[2] Vernon Austel. Implementing abstract types in ACL2. In *ACL2 Workshop 2003*, July 2003.

[3] Mike Fagan. *Soft Typing: An approach to type checking for dynamically typed languages.* PhD thesis, University of Texas, Houston, 1992.

[4] Steven L. Jenkins and Gary T. Leavens. Polymorphic type-checking in Scheme. Technical Report 91-21a, Iowa State University, 1996.

[5] Andreas Kind and Horst Friedrich. A practical approach to type inference for EuLisp. *Lisp and Symbolic Computation*, 1993.

[6] Christian Lindig. STYLE - a practical type checker for Scheme. Master's thesis, Technische Universität Braunschweig, 1993.

[7] Manfred Widera and Christoph Beierle. Combining strict and soft typing in functional programming. In K. Beiersdrfer, G. Engels, and W. Schfer, editors, *Informatik'99 – Informatik berwindet Grenzen*, pages 350–359. Springer-Verlag, 1999.

[8] Andrew K. Wright. *Practical Soft Typing*. PhD thesis, University of Texas, Houston, 1994.