Fix Your Types

Sol Swords Jared Davis
Centaur Technology, Inc.
7600-C N. Capital of Texas Hwy, Suite 300
Austin, TX 78731
{sswords,jared}@centtech.com

When using existing ACL2 datatype frameworks, many theorems require type hypotheses. These hypotheses slow down the theorem prover, are tedious to write, and are easy to forget. We describe a principled approach to types that provides strong type safety and execution efficiency while avoiding type hypotheses, and we present a library that automates this approach. Using this approach, types help you catch programming errors and then get out of the way of theorem proving.

1 Introduction

ACL2 is often described as "untyped," and this is certainly true to some degree. Terms like `(+ 0 "hello")`, which would be not be accepted by the static type checks of languages like Java, are legal and well-defined in the logic, and can even be executed so long as guard checking is disabled. Terms like `(/ 5 0)`, which would be well-typed but would cause a run-time error in most programming languages, are also logically well-defined and can also be executed when guards are not checked.

Of course, most ACL2 code is written with particular types in mind, often expressed as the guards of the functions. When proving properties of such code, it's easy to get tripped up by corner cases where some variables of the theorem are of the wrong types. To avoid this, one strategy is to begin a theorem with a list of type hypotheses, one for each variable mentioned. These hypotheses act as a kind of insurance: we may not know whether they're necessary or not, but including them might save us from having to debug failed proofs caused by missing type assumptions.

On the other hand, lists of type hypotheses are often repetitive, take time to write, and make the formulas we're proving larger and less elegant. We can hide much of this with macros, e.g., the `define` utility has special options like `:hyp :guard` for including the guard as a hypothesis in return-value theorems. But even then, these hypotheses will cause extra work for the rewriter since it must relieve them before it can apply our theorem, and may make it harder to carry out later proofs since our theorem will not be applied unless its hypotheses can be relieved.

Accordingly, after we have proven a theorem, a good practice is to try to "strengthen" it by removing any unnecessary hypotheses. There is even a tool, `remove-hyps`, that tries to automatically identify unnecessary hypotheses ex post facto. While strengthening theorems is useful, it is limited. For instance, we (of course) cannot eliminate type hypotheses that are actually necessary for the formula to be a theorem. It can also be tedious, e.g., automation such as `define`'s `:hyp :guard` does not provide a convenient way to remove parts of the guard. This is not a purely theoretical concern; see for instance ACL2 Issue 167, a request for such a feature.

An alternate strategy, which is well-known and certainly not novel, is to more carefully code our functions so that they always treat ill-typed inputs according to some particular *fixing convention*. By following this approach, we can typically avoid the need for type hypotheses altogether. Many examples of this approach can be found throughout ACL2. To name a few:

Submitted to:
ACL2 2015

- Arithmetic functions treat non-numbers as 0; functions expecting a particular type of number (integer, natural) treat anything else as 0—e.g., `zp`, `nth`, `logbitp`.
- Functions expecting a string treat a non-string as `""`—e.g., `char`, `(coerce x 'list)`.
- Any atom is treated as `nil` by `car`, `cdr`, `endp`, etc.
- The `std/osets` [5] library functions treat non-sets as `nil`.

Following this strategy typically takes a small bit of initial setup, e.g., to agree upon and implement a fixing convention. But once this convention is in place, type hypotheses can be eliminated from many theorems. For instance, we have unconditional theorems such as $a + b = b + a$ without hypotheses about a and b being numbers, and $X \subseteq X$ without hypotheses about X being a set. By eliminating type hypotheses, these theorems become easier to read and write, and can be more efficiently and reliably used to simplifying later proof goals.

Unfortunately, existing datatype definition frameworks for ACL2 don't provide any easy way to follow the fixing strategy. For example, consider the available macros for introducing product types, like `defstructure` [3], `defdata` [4], and `defaggregate`. These macros define constructors and accessor functions that do not support any particular convention for dealing with ill-typed fields or products. Consider a simple `student` structure introduced by:

```
(defaggregate student
  ((name stringp)
   (age  natp)))
```

The constructor and accessors for `student` structures will have strong guards that are useful for revealing programming errors in function definitions. However, in the logic, nothing prevents us from invoking the `student` constructor on ill-typed arguments. For instance, in cases like:

```
(make-student :name 6 :age "Calista")
```

the constructor fails to produce a valid `student-p`. The accessors suffer from similar problems, for instance the following term is equal to `6`, which is not a well-typed student name:

```
(student->name (make-student :name 6 :age "Calista"))
```

Consequently, reasoning about these structures almost always requires type hypotheses. Since the types defined by these frameworks are often found at the lowest levels of ACL2 models, these hypotheses percolate upwards, infecting the entire the code base!

In this paper, we address this problem with the following contributions:

- We present, in precise terms, a **fixtype discipline** for working with types in ACL2 (Section 2). This discipline allows efficient reasoning via avoiding type hypotheses, strong type checking via ACL2's guard mechanism, and preserves efficient execution via `mbe`.
- Manually following the fixtype discipline would be tedious. Accordingly, we present a new library, `FTY` (short for "fixtypes"), which provides automation for following the discipline (Section 3). The FTY library contains tools that automate the introduction of new types and assist with creating functions that "properly" operate on those types.

While there is room for improvement (Section 4), the approach and automation that we present is practical and scales up to complex modeling efforts. We have successfully used FTY as the type system for two large libraries: VL, which processes Verilog and SystemVerilog source code; and SV, a hardware modeling and analysis framework. VL, in particular, involves a very complex hierarchy of types. For instance, it includes a 30-way mutually recursive datatype that represents SystemVerilog expressions, types, and related syntactic constructs.

2 The Fixtype Discipline

We begin, in this section, by describing in precise terms a *fixtype discipline* for working with types in ACL2. In our experience, following this discipline is an effective way to obtain the benefits of strong type checking while keeping types out of the way of theorem proving.

The basic philosophy behind the fixtype discipline is that all functions that take inputs of a particular type should treat any inputs that are *not* of that type in a consistent way. This can be done using *fixing functions*.

Definition 1. A **fixing function** *fix* for a (unary) type predicate *typep* is a (unary) function that (1) always produces an object of that type, and (2) is the identity on any object of that type. That is, it satisfies:

$$(1) \quad \forall x : typep(fix(x))$$
$$(2) \quad \forall x : typep(x) \Rightarrow fix(x) = x$$

Given a fixing function, an easy way to ensure that some new definition treats all of its inputs in a type-consistent way is to immediately apply the appropriate fixing function to each input before proceeding with the main body of the function. For guard-verified functions, this preliminary fixing can be done for free using `mbe`. Alternatively, if all occurrences of an input variable in the function's body occur in contexts that are already type-consistent, then explicit fixing isn't necessary.

When a function follows this approach, the fixing functions become "transparent" to that function. For instance, since `nth` properly fixes its index argument to a natural number, the following holds:

```
(defthm nth-of-nfix
  (equal (nth (nfix n) x)
         (nth n x)))
```

Given any fixing function, we can define a corresponding equivalence relation. For instance, for naturals, we can define `nat-equiv` as equality up to `nfix`:

```
(defun nat-equiv (x y)
  (equal (nfix x) (nfix y)))
```

Functions that properly fix their arguments will satisfy a congruence for this equivalence under equality: that is, they produce `equal` results when given equivalent arguments. For instance, for `nth`:

```
(defthm nat-equiv-congruence-for-nth
   (implies (nat-equiv n m)
            (equal (nth n x)
                   (nth m x)))
  :rule-classes :congruence)
```

We can now define our fixtype discipline.

Definition 2. A function follows the **fixtype discipline** if, for each typed input, the type has a corresponding fixing function and equivalence relation, and the function produces equal results given type-equivalent inputs.

A consequence of following the fixtype discipline is that theorems can avoid type hypotheses.

Theorem 1. *Let typep be a type and let $\equiv$ be the equivalence relation induced by a fixing function for typep. Let $C(x)$ be a conjecture satisfying the congruence*

$$(x \equiv x') \Rightarrow \left(C(x) \Leftrightarrow C\left(x'\right)\right).$$

Then $C(x)$ is a theorem if and only if $typep(x) \Rightarrow C(x)$ is a theorem.

This congruence means that $C(x)$ is a formula where the variable x is consistently treated according to the fixing discipline for *typep*; in this case, it isn't necessary to include $typep(x)$ as a hypothesis. This generalizes easily to additional variables.

3 The FTY Library

If we want to follow the fixtype discipline, all of our type predicates need to have corresponding fixing functions and equivalence relations. Also, as we introduce new functions that operate on these types, we need to prove that these functions satisfy the appropriate congruences, i.e., that they treat their inputs in a type-consistent way.

Although these definitions and proofs are straightforward, it would be very tedious to carry them out manually. It would also be difficult to make use of libraries like `std/util` or `defdata` since the functions these frameworks introduce do not follow the fixtype discipline. This is unfortunate because these libraries really make it far more convenient to introduce new types.

To address this, we have developed a new library, named FTY, which provides several utilities to automate this boilerplate work and to facilitate the introduction of new types that follow the discipline. Among these utilities, we have:

- `deffixtype`, which associates a type predicate with a fixing function and equivalence relation, for defining base types like `natp`, `stringp`, and custom user-defined base types. (Section 3.1)
- `deftypes` and associated utilities `defprod`, `deftagsum`, `deflist`, `defalist`, and more, which define new derived fixtype-compliant product types, sum types, list types, etc. (Section 3.2)
- `deffixequiv` and `deffixequiv-mutual`, which prove the appropriate type congruences for functions that operate on these types. (Section 3.3)

We now briefly describe these utilities. We focus here on what these utilities automate and how this helps to make it easier to follow the fixtype discipline. More detailed information on how to practically make use of these utilities, their available options, etc., can be found in the FTY documentation [8] in the ACL2+Books Manual.

3.1 Deffixtype

The FTY library uses an ACL2 `table` to record the associations between the name, predicate, fixing function, and equivalence relation for each known type. This information is used by many later FTY utilities to improve automation. For instance, when we define a new structure, this table allows us to look up the right fixing function and equivalence relation to use for each field just by its type, without needing to be repetitively told the fixing function and equivalence relation for every field.

The **deffixtype** utility is used to register new *base types*, i.e., types that are not defined in terms of other FTY types, with this table. Here is an example, which registers a new type named `nat`, recognized by `natp`, with fixing function `nfix`, and with the equivalence relation `nat-equiv`:

```
(deffixtype nat
  :pred  natp
  :fix   nfix
  :equiv nat-equiv)
```

The type name does not need to be a function name; we typically use the name of the predicate without the final "p" or "-p." The predicate and fixing function must always be provided by the user and defined ahead of time. `Deffixtype` can automatically define the equivalence relation based on the fixing function, or it can use an existing equivalence relation.

We usually do not need to invoke `deffixtype` directly. FTY includes a `basetypes` book that sets up these associations for basic ACL2 types like naturals, integers, characters, Booleans, strings, etc. When new derived types are introduced by FTY macros like `deftypes` (Section 3.2), they are automatically registered with the table. On the other hand, `deffixtype` is occasionally useful for defining low-level custom base types, or types that use special encodings, or that for some other reason we prefer not to introduce with `deftypes`.

Choosing a good fixing function for a type is not always straightforward. As far as the fixtype discipline and the FTY library is concerned, any function that satisfies Definition 1 suffices. However, the way in which ill-typed objects are mapped into the type affects which functions will have proper congruences for the induced equivalence relation. Some choices are dictated by pre-existing ACL2 conventions; for example, if we wrote our own `my-nat-fix` function that coerced non-naturals to 5 instead of 0, then this fixing function wouldn't be transparent to built-in functions such as `zp` and `nth`.

The fixing function's guard may optionally require that the input object already be of the type. This allows the fixing function to be coded so that it is essentially free to execute, using `mbe` so that the executable body is just the identity. It is also generally useful to inline the fixing function to avoid the small overhead of a function call. For example:

```
(defun-inline string-fix (x)
  (declare (xargs :guard (stringp x)))
  (mbe :logic (if (stringp x) x "")
       :exec x))
```

3.2 Deftypes and Supporting Utilities

Whereas `deffixtype` is useful for registering base types and special, custom types, the **deftypes** suite of tools can be used to easily define common kinds of derived types. The constructors, accessors, and other supporting functions introduced for these types follow the fixtype discipline, and the new types are automatically registered with `deffixtype`. There are utilities for introducing many kinds of types:

- `defprod`, which defines a product type,
- `deftagsum`, which defines a tagged sum of products,
- `deflist`, which defines a list type which has elements of a given type,
- `defalist`, which defines an alist type with keys and values of given types,
- `defoption`, which defines an option/maybe type,
- and others.

Using these macros is not much different than using other data definition libraries. For instance, we can introduce a basic `student` structure as follows:

```
(defprod student
  ((name stringp)
   (age  natp)))
```

This is very much like introducting a structure with `defaggregate`: it produces a recognizer, constructor, accessors for the fields, `b*` binders, and readable make/change macros. Unlike `defaggregate`, it also generates a fixing function, `student-fix`, an equivalence relation, `student-equiv`, and registers the new student type with `deffixtype`. The constructor and accessor functions for the new type also follow the fixtype discipline, e.g., we unconditionally have theorems such as:

- `(student-p (student name age))`
- `(stringp (student->name x))`
- `(natp (student->age x))`

A notable feature of `deftypes` is that it also provides strong support for mutually recursive types. In particular, several calls of utilities such as `defprod`, `deflist`, etc., may be combined inside a `deftypes` form to create a mutually-recursive clique of types. For example, to model a simple arithmetic term language such as:

$$\begin{array}{rcl} \text{aterm} & = & \mathsf{Num}\ \{\texttt{val} :: \text{integer}\} \\ & | & \mathsf{Sum}\ \{\texttt{args} :: \text{List aterm}\} \\ & | & \mathsf{Minus}\ \{\texttt{arg} :: \text{aterm}\} \end{array}$$

We might write the following `deftypes` form:

```
(deftypes arithmetic-terms
  (deftagsum aterm
    (:num   ((val integerp)))
    (:sum   ((args atermlist)))
    (:minus ((arg aterm))))
  (deflist atermlist
    :elt-type aterm))
```

As you might expect, this form creates the basic predicates, fixing functions, and equivalence relations for `aterm`s that are needed for the fixtype discipline:

- Predicates `aterm-p` and `atermlist-p`,
- Fixing functions `aterm-fix` and `atermlist-fix`, and
- Equivalence relations `aterm-equiv` and `atermlist-equiv`.

It also registers the new types with `deffixtype`. The form also defines several functions and tools for working with these new types, all of which have appropriate congruences for the fixtype discipline:

- A kind function, `aterm-kind`, to determine the kind of an `aterm`, e.g., `:num`, `:sum`, or `:minus`.
- Constructors for each kind of `aterm`: `aterm-num`, `aterm-sum`, and `aterm-minus`, and associated make/change macros in the style of `defaggregate`/`defprod`.
- Accessors for each kind of `aterm`: `aterm-num->val`, `aterm-sum->args`, `aterm-minus->arg` and associated `b*` binders.

- Measure functions, `aterm-count` and `atermlist-count`, appropriate for structurally recurring over objects of these types.

For convenience, a macro `aterm-case` is also introduced. This macro allows us to implement the common coding scheme of cases on the kind of an `aterm`, followed by binding variables to any needed fields of the product. Here is a simple example of using `aterm` structures.

```
(defines aterm-eval
  (define aterm-eval ((x aterm-p))
    :measure (aterm-count x)
    :returns (val integerp)
    :verify-guards nil
    (aterm-case x
      :num x.val
      :sum (atermlist-sum x.args)
      :minus (- (aterm-eval x.arg))))

  (define atermlist-sum ((x atermlist-p))
    :measure (atermlist-count x)
    :returns (val integerp)
    (if (atom x)
        0
      (+ (aterm-eval (car x))
         (atermlist-sum (cdr x)))))
  ///
  (verify-guards aterm-eval))
```

3.3 Deffixequiv and Deffixequiv-mutual

Together, `deffixtype` and `deftypes` allow us to largely automate the process of introducing new types that support the fixtype discipline. But this is only half the battle. When we define new functions that make use of these types, we are still left with the task of proving that these functions satisfy the appropriate congruences for every argument of these types. If our model or program involves many function definitions, this can be a lot of tedious work.

To automate this process, FTY offers two related utilities, **deffixequiv** and **deffixequiv-mutual**. These utilities are integrated with `define` and `defines` and also make use of the table of types from `deffixtype`. This allows them to figure out what theorems are needed, often without any help at all. In particular, the types of the arguments are inferred from the extended formals of the each function. The corresponding fixing functions and equivalence relations can then be looked up from the table, and the appropriate congruence rules can be generated. Besides congruence rules, we additionally generate rules that normalize constant arguments to their type-fixed forms.

Consider the `aterm-eval` example above. To generate the congruence rules for both `aterm-eval` and `atermlist-eval`, it suffices to invoke:

```
(deffixequiv-mutual aterm-eval)
```

The `deffixequiv-mutual` macro determines the types of the arguments by examining the guards specified in the `define` formals, and it uses the flag induction scheme produced by `defines` to automatically prove the congruence.

For recursive or mutually-recursive functions, proving a congruence directly can be difficult because there are two calls of the function in the statement of the theorem, and these two calls may suggest different induction schemes that may not be simple to merge. However, the congruences we are concerned with follow from the fact that the fixing function is transparent to the function, which can usually be proved straightforwardly by induction on the function's own recursion scheme. In practice, the `deffixequiv` and `deffixequiv-mutual` utilities usually fully automate the derivation of the congruence from the transparency theorem.

Even if we only need to write a `deffixequiv` or `deffixequiv-mutual` form after each definition, this can be easy to forget. To further automate following the discipline, you can optionally enable a *post-define hook* that will automatically issue a suitable `deffixequiv` command after each definition. See the documentation for `fixequiv-hook` for details.

4 Challenges and Future Work

The FTY library provides a robust implementation of a type system that would feel familiar to users of strongly typed functional programming languages such as Haskell or ML. However, there are a few pitfalls in their practical use, which we discuss below along with potential solutions.

4.1 Generic Functions

The most common problem in working with the fixtype discipline is in the use of generic functions such as `assoc`, `append`, and many others. These functions are designed to work on objects of nonspecific type, and therefore don't follow fixing conventions for specific types. One can always apply appropriate fixing functions to the inputs of these functions, so programming with them in a fixtype discipline isn't hard. However, applying this simple strategy to theorems will often result in ineffective rewrite rules.

For example, suppose `(bind-square-to-root key alist)` fixes `key` to type `natp` and `alist` to type `nat-nat-alist-p`, and we want to prove a theorem like the following:

```
(equal (assoc k (bind-square-to-root k rest))
       (or (and (square-p k)
                (cons k (nat-sqrt k)))
           (assoc k rest)))
```

Presumably this isn't true without some type assumptions. One way to fix the theorem is to apply fixing functions everywhere that typed variables are used in generic contexts:

```
(equal (assoc (nfix k) (bind-square-to-root k rest))
       (or (and (square-p k)
                (cons (nfix k) (nat-sqrt k)))
           (assoc (nfix k) (nat-nat-alist-fix rest))))
```

But this rewrite rule is not always applicable. The left hand side will match only when we have an explicit `nfix` in our goal, but this `nfix` is likely to be simplfied away in cases where the key is known to be a natural. In these cases, a formulation with a type hypothesis would work:

```
(implies (natp k)
         (equal (assoc k (bind-square-to-root k rest))
                ...))
```

Unfortunately, this rule typically won't allow us to simplify terms such as:

```
(assoc (nfix k) (bind-square-to-root k rest))
```

because it fails to unify. In general, both kinds of terms may be encountered and both formulations of the rule may be useful. To solve this problem, one might consider automation to generate both forms of the theorem, or to generate a theorem that catches both cases as follows:

```
(implies (and (syntaxp (or (equal k1 k) (equal k1 `(nfix ,k))))
              (nat-equiv k1 k)
              (natp k1))
         (equal (assoc k1 (bind-square-to-root k rest))
                ...))
```

For the moment, unfortunately, reasoning about a mix of generic functions with fixtype-discipline functions seems to require the sort of consideration of types that we had hoped to avoid. We generally deal with these problems on an ad-hoc basis, either by proving both forms of the theorem or, in more problematic cases, by introducing typed alternatives to the generic functions involved.

4.2 Subtypes

It is possible to use the FTY library to define two types that have a subtype relation, but the library doesn't have any automation for proving or making use of this relationship.

In practice, we have found it difficult to get subtype relations to work well. Proving theorems about a mixture of functions that operate on sub- and supertypes has the same problems as proving theorems with a mixture of generic and fixtype functions, as discussed above. Reasoning about a subtype hierarchy also can lead to degraded prover performance, since proving that something is of type A may lead by backchaining to attempting to prove it to be of each subtype of A.

4.3 Parameterized Types

Haskell and ML support types that take other types as parameters, e.g., List A signifying a list of objects of type A, where A is a type variable. Function signatures may contain types that are not fully specified, and these functions may later be used in contexts where the type variables are concretized as particular types.

Selfridge and Smith [17] created a macro library that supports a form of polymorphism by automating the creation of instances of the `defsum` macro. Polymorphic functions are then supported by another set of macros that allow one to instantiate a template function definition with different substitutions for type variables. A similar macro library could be used to add polymorphism via templates to FTY, but this has not yet been done.

4.4 Dependent Types

Correct behavior of multiple-input functions often depends on constraints involving more than one of the inputs. The fixtypes discipline is focused on unary types, but occasionally it is desirable for a product type to contain multiple elements that have constraints linking them. We have experimentally implemented support for this in `defprod` and `deftagsum` by allowing the user to specify these constraints along with an extra fixing step that forces the fields to satisfy these constraints; this works in practice for simple constraints like "a literal's value should fit into its width." We expect that there would be difficulties in formulating constraints between subfields of a recursive data structure.

4.5 Symbolic and Logical Evaluation

Execution efficiency of functions using the fixtype discipline is highly dependent on the use of `mbe` to avoid calling fixing functions. Evaluation in the logic (with guard checking turned off) is much more expensive with such functions because the `:logic` part of the `mbe` then needs to be executed.

This problem also applies to symbolic evaluation with the GL system [18]. GL is used in hardware verification at Centaur and elsewhere; it evaluates ACL2 functions on bit-level symbolic inputs, producing bit-level symbolic results, allowing the use of SAT or BDD reasoning to prove ACL2 theorems. However, GL ignores guards (except for concrete evaluation) and instead symbolically simulates the logical definitions of functions. Therefore, when using GL on fixtype-compliant functions, these fixing functions will be unnecessarily (symbolically) executed frequently.

This extra expense could be problematic in some cases. In future work we expect to address this by adding a facility to FTY to generate extra GL rules to help it avoid executing fixing functions. Currently, we have worked around this problem in some cases by using fixing functions that are cheap to symbolically execute. For example, if we are dealing with, say, a 32-bit unsigned integer type, then for symbolic simulation it is cheaper to use `(loghead 32 x)` rather than `(if (unsigned-byte-p 32 x) x 0)` as the fixing function. This expense of fixing can also be avoided by creating custom symbolic counterparts, which are used in important core routines in hardware verification frameworks like `ESIM` and `SV`.

4.6 Traversal of Complicated Data Structures

In languages like ML or Haskell, it is possible to write higher order functions for traversing deeply nested data structures. This capability goes a long way toward making it reasonable to inspect and manipulate such objects. Since ACL2 is first order, we cannot write these kinds of generic traversals. Instead, we have to duplicate the boilerplate code for traversing a structure in each algorithm that operates on it. This can become very tedious.

For example, the parsetree format for the VL Verilog/SystemVerilog toolkit contains 168 datatypes, 132 of which are defined in terms of other types (as a product, list, etc.), reflecting the complexity of the SystemVerilog language. We might like to, for instance, collect all identifiers used in a module. We might also like simplify all expressions throughout a module. Doing either of these will require traversing many of the same structures (modules, declarations, assignments, etc.) to reach the objects of interest (identifiers, expressions).

We have implemented an experimental utility, `defvisitor`, intended to generate the boilerplate code necessary for these situations. The user provides code to be run on certain types and to combine results from recursive calls, and the utility generates the boilerplate to traverse the data structures. The current implementation is a proof of concept and its user interface is likely to change, but it has been used to implement several algorithms within the VL library.

5 Related Work

5.1 Fixing Conventions

The use of fixing conventions to avoid hypotheses is well studied and has been used since the earliest Boyer Moore provers. In Boyer and Moore's 1979 *A Computational Logic* we find a `fix` function for NQTHM's naturals and hypothesis-free theorems such as the commutativity of plus. Boyer and Moore

credit A. P. Morse as the inspiriation for this approach, citing his treatment of set theory, *A Theory of Sets* [16], and recalling[1] that:

> "Morse tried every way he could to fix every function that he introduced to eliminate hypotheses if possible, without doing any damage. He delighted in such theorems as that *and* and *or* distributed over one another for all arguments, no matter what objects the arguments were, no matter that *and* was the exact same as set intersection and that *or* was the same as set union."

In their 1994 *Design Goals for ACL2*, Kaufmann and Moore reflect that "NQTHM's logic gets incredible mileage out of the notion that functions—especially arithmetic functions—default 'unexpected' inputs to reasonable values so that many theorems are stated without hypotheses." As a result, fixing conventions were used liberally in their new theorem prover, e.g., throughout its completion axioms for primitive functions on numbers, characters, strings, etc.

Since then, many ACL2 libraries such as `std/osets` [5] and `bitops`, have made heavy use of the technique. Perhaps the most extreme examples are found in `misc/records` [14] and related work such as typed records [11], `defexec`-enhanded records [12], memories [6], and `defrstobj`, which each use sophisticated, convoluted fixing functions to achieve hypothesis-free read-over-write theorems.

Fixing conventions are often unnecessary in typed logics. In such a logic, when we define functions such as PLUS : NAT × NAT → NAT, there is no need to include any type hypotheses in theorems such as the commutativity of PLUS, because any attempt to call or reason about PLUS on non-NAT arguments is simply an error. On the other hand, even in such a logic, fixing conventions may be useful for modeling behavior of operations whose intended domains are not easy to describe using types. Lamport and Paulson [15] provide an engaging discussion of these sorts of issues.

5.2 Data Struture Libraries

There has been significant previous work to develop data structure libraries for ACL2. An early example is Brock's classic `data-structures` library, which featured macros such as `defstructure` [3]. As a concrete example of using this macro, we might write:

```
(defstructure student
  (name (:assert (stringp name) :type-prescription))
  (age  (:assert (natp age) :type-prescription)))
```

This produces a constructor that simply conses together its arguments and accessors that simply `car`/`cdr` into their argument. No fixing is done, so the constructor only produces a well-formed `student-p` if its arguments have the proper types, and the accessors may produce ill-typed results when applied to non-`student-p` objects. Accordingly, reasoning about such structures typically requires type hypotheses. The more recent `defaggregate` macro follows this same approach.

ACL2's single threaded objects [2] are in many ways similar to `defstructure` and `defaggregate`. Although it probably would make little sense to define a student structure as a stobj, we can do so:

```
(defstobj student
  (name :type string        :initially "")
  (age  :type (integer 0 *) :initially 0))
```

[1] Correspondence with Bob Boyer and J Moore.

The resulting recognizer, accessors, and mutators are similar to those produced by `defstructure` or `defaggregate` and so reasoning about these operations usually requires type hypotheses. On the other hand, the recent addition of abstract stobjs [10] makes it possible to develop alternative logical interfaces, e.g., we could arrange so that the `student` stobj was logically viewed as an FTY product object.

The `defdata` [4] library by Chamarthi, Dillinger, and Manolios features an alternative macro, also called `defdata`, that supports introducing richer types, such as sum types, mutually recursive types, etc. This framework also features integrated support for counterexample generation, an exciting feature which FTY does not yet have. To define a similar `student` structure with `defdata` we might write:

```
(defdata student (record (name . string)
                         (age  . nat)))
```

This similarly results in a `studentp` recognizer, constructor, and accessors like `student-age`. Unlike `defstructure` or `defaggregate`, the underlying representation of these functions is based on an optimized variant [12] of Kaufmann and Sumners' records book [14], and the macro provides special integration with ACL2's `Tau` reasoning procedure. However, the general approach to type reasoning about these structures is unchanged: we still require type hypotheses to establish that the construct produces a valid `studentp`, that `student-name` returns a string, and so forth.

Despite type hypotheses, macros like `defstructure`, `defaggregate`, and `defdata` are certainly very useful. `Defstructure` has long been used in ACL2 developments, including recent work such as the modeling by Hardin, et al. [13] of the LLVM compiler project's intermediate form in ACL2, and the formalization by van Gastel and Schmaltz [9] of the xMAS language for communication networks on multi-core processors and systems-on-chip. For many years, we used `defaggregate` and other `std/util` macros at Centaur as the basis for our VL library, microcode model [7], and other internal applications. In our experience, porting these libraries to FTY was not difficult and has helped to simplify our code.

5.3 Make-Event Metaprogramming

In 2004, Vernon Austel developed [1] an experimental variant of ACL2 that added support for a certain type system. He explained that this work had required modifying ACL2 because "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," and proposed extending ACL2 with something like `make-event` to "allow others to experiment with type systems without having to hack the system code." Indeed, our FTY library makes extensive use of `make-event` to record the associations between type recognizers, fixing functions, and equivalence relations, and to look up (via `define`) the type signatures for functions.

6 Conclusion

FTY is a new data structure library for ACL2 that provides deep support for using fixing functions to avoid type hypotheses in theorems. Its successful use may require somewhat more discipline than similar libraries such as `std/util` or `defdata`. In exchange, it provides a strongly typed programming environment that can help to catch errors during development while largely avoiding type hypotheses during theorem proving.

Having a good data structure library is tremendously useful when developing large systems in ACL2. A fixing discipline is one part of this, but FTY is also increasingly mature and capable in other ways,

e.g., it retains much of the `std/util` look and feel, with features such as `XDOC` integration, convenient `b*` binders, readable `make/change` macros, etc. We are now using FTY as for large ACL2 libraries such as SV and VL libraries, and have been pleased with the results.

The source code for FTY is included in the ACL2 Community Books under the `centaur/fty` directory. Beyond this paper, the FTY library has extensive documentation, which includes more detailed information on the available options for each macro. The `centaur/fty` directory also includes various test cases that may serve as useful examples of using the library.

We hope you find FTY useful.

6.1 Acknowledgments

We thank Bob Boyer and J Moore for very interesting discussions about the origins of fixing disciplines in Boyer-Moore provers. We thank Shilpi Goel and Cuong Chau for their corrections and feedback on this paper.

References

[1] Vernon Austel (2004): *Adding a typing mechanism to ACL2*. ACL2 '04. Available at `http://www.cs.utexas.edu/users/moore/acl2/workshop-2004/contrib/austel/acl2-types.pdf`.

[2] Robert S. Boyer & J Strother Moore (2002): *Single-Threaded Objects in ACL2*. In: *Practical Aspects of Declarative Languages*, *LNCS* 2257, Springer, pp. 9–27, doi:10.1007/3-540-45587-6_3.

[3] Bishop Brock (1997): *Defstructure for ACL2*. Available at `http://www.cs.utexas.edu/users/moore/publications/others/defstructure-brock.ps`.

[4] Harsh Raju Chamarthi, Peter C. Dillinger & Panagiotis Manolios (2014): *Data Definitions in the ACL2 Sedan*. In: *ACL2 '14*, EPTCS, pp. 27–48, doi:10.4204/EPTCS.152.3.

[5] Jared Davis (2004): *Finite Set Theory based on Fully Ordered Lists*. ACL2 '04. Available at `http://www.cs.utexas.edu/users/moore/acl2/workshop-2004/contrib/davis/set-theory.pdf`.

[6] Jared Davis (2006): *Memories: Array-like records for ACL2*. In: *ACL2 '06*, ACM, pp. 57–60, doi:10.1145/1217975.1217986.

[7] Jared Davis, Anna Slobodova & Sol Swords (2014): *Microcode Verification: Another Piece of the Microprocessor Verification Puzzle*. In: *ITP '14*, *LNCS* 8558, Springer, pp. 1–16, doi:10.1007/978-3-319-08970-6_1.

[8] Jared Davis & Sol Swords (2015): *XDOC Documentation for FTY*. Available at `http://www.cs.utexas.edu/users/moore/acl2/manuals/`.

[9] Bernard van Gastel & Julien Schmaltz (2013): *A formalisation of XMAS*. In: *ACL2 '13*, EPTCS, pp. 111–126, doi:10.4204/EPTCS.114.9.

[10] Shilpi Goel, Warren A. Hunt, Jr. & Matt Kaufmann (2013): *Abstract Stobjs and their application to ISA modeling*. In: *ACL2 '13*, EPTCS, pp. 54–69, doi:10.4204/EPTCS.114.5.

[11] David Greve & Matthew Wilding (2003): *Typed ACL2 Records*. ACL2 '03. Available at `http://www.cs.utexas.edu/users/moore/acl2/workshop-2003/contrib/greve-wilding_defrecord/defrecord.pdf`.

[12] David A. Greve, Matt Kaufmann, Panagiotis Manolios, J Strother Moore, Sandip Ray, José Luis Ruiz-Reina, Rob Sumners, Daron Vroon & Matthew Wilding (2008): *Efficient execution in an automated reasoning environment*. *Journal of Functional Programming* 18, pp. 15–46, doi:10.1017/S0956796807006338.

[13] David S. Hardin, Jennifer A. Davis, David A. Greve & Jedidiah R. McClurg (2014): *Development of a Translator from LLVM to ACL2*. In: *ACL2 '14*, EPTCS, pp. 163–177, doi:10.4204/EPTCS.152.13.

[14] Matt Kaufmann & Rob Sumners (2002): *Efficient Rewriting of Data Structures in ACL2*. ACL2 '02. Available at `http://www.cs.utexas.edu/users/moore/acl2/workshop-2002/contrib/kaufmann-sumners/rcd.pdf`.

[15] Leslie Lamport & Lawrence C. Paulson (1999): *Should Your Specification Language Be Typed*. *ACM Transactions on Programming Languages and Systems* 21(3), pp. 502–526, doi:10.1145/319301.319317.

[16] Anthony P. Morse (1965): *A Theory of Sets*. Academic Press.

[17] Benjamin Selfridge & Eric Smith (2014): *Polymorphic Types in ACL2*. In: *ACL2 '14*, EPTCS, pp. 49–60, doi:10.4204/EPTCS.152.4.

[18] Sol Swords & Jared Davis (2011): *Bit-Blasting ACL2 Theorems*. In: *ACL2 '11*, *Electronic Proceedings in Theoretical Computer Science* 70, pp. 84–102, doi:10.4204/EPTCS.70.7.