Isomorphic Data Type Transformations

Alessandro Coglio
Kestrel Institute
http://www.kestrel.edu

Stephen Westfold
Kestrel Institute
http://www.kestrel.edu

In stepwise derivations of programs from specifications, data type refinements are common. Many data type refinements involve isomorphic mappings between the more abstract and more concrete data representations. Examples include refinement of finite sets to duplicate-free ordered lists or to bit vectors, adding record components that are functions of the other fields to avoid expensive recomputation, etc. This paper describes the APT (Automated Program Transformations) tools to carry out isomorphic data type refinements in the ACL2 theorem prover and gives examples of their use. Because of the inherent symmetry of isomorphisms, these tools are also useful to verify existing programs, by turning more concrete data representations into more abstract ones to ease verification. Typically, a data type will have relatively few interface functions that access the internals of the type. Once versions of these interface functions have been derived that work on the isomorphic type, higher-level functions can be derived simply by substituting the old functions for the new ones. We have implemented the APT transformations `isodata` to generate the former, and `propagate-iso` for generating the latter functions as well as theorems about the generated functions from the theorems about the original functions. `Propagate-iso` also handles cases where the type is a component of a more complex one such as a list of the type or a record that has a field of the type: the isomorphism on the component type is automatically lifted to an isomorphism on the more complex type. As with all APT transformations, `isodata` and `propagate-iso` generate proofs of the relationship of the transformed functions to the originals.

1 Background, Motivation, and Contribution

In stepwise program refinement [10, 24], an implementation is derived from a specification via a sequence of intermediate specifications. A derivation is a sequence $s_0 \rightsquigarrow s_1 \rightsquigarrow \ldots \rightsquigarrow s_n \overset{\mathbf{c}}{\mapsto} p$, where: s_0 is a requirements specification in some specification language; $s_1, \ldots, s_n$ are intermediate specifications in the same language; $\rightsquigarrow$ is a refinement relation (a preorder, i.e. reflexive and transitive); and p is an implementation in some programming language, obtained from s_n via a code generator $\overset{\mathbf{c}}{\mapsto}$. If the specification language is a superset of a programming language, code generation may be omitted and $p = s_n$.

If every derivation step (including code generation if applicable) is formally verified, the implementation is provably correct by construction. Given s_i, the step $s_i \rightsquigarrow s_{i+1}$ may be realized in two ways. One is by writing s_{i+1} and verifying the $\rightsquigarrow$ relation; this is 'posit-and-prove' [1, 22, 13]. The other (when possible) is by automatically generating both s_{i+1} and the $\rightsquigarrow$ proof from s_i via an *automated transformation* [19, 21, 16, 7, 14]. The latter approach may require the developer to prove *applicability conditions*, i.e. theorems from which $\rightsquigarrow$ can then be proved automatically; but proving applicability conditions is generally easier than proving $\rightsquigarrow$. Code generation $s_n \overset{\mathbf{c}}{\mapsto} p$ is normally automatic; its verification is akin to compilation verification.

Each step $s_i \rightsquigarrow s_{i+1}$ narrows down the possible implementations or rephrases the specification towards an implementation, e.g. by choosing an algorithm or data structure, or by exploiting algebraic laws to optimize expressions. A common case is that of a data type refinement, where a more abstract data representation is turned into a more concrete one [12]. A large and interesting subset of data type refinements

To appear in EPTCS.

involves isomorphic mappings between abstract and concrete representations; examples are refining finite sets to duplicate-free ordered lists or to bit vectors, adding record components that are computable from the others for caching, and some perhaps less expected ones as in Section 4.2. Many other data type refinements do not involve isomorphic mappings; see Section 6.

In an interactive theorem prover like ACL2, stepwise program refinement can be carried out using predicates over (deeply or shallowly embedded) implementations as specifications s_i and (reversed) implication as $\rightsquigarrow$ (an approach called 'pop-refinement') [4, 5]; and $\overset{\mathbf{c}}{\mapsto}$ may be realized either as a $\rightsquigarrow$ sequence entirely in the logic of the prover [4], or via a more typical code generator [6] [23, Topic ATJ]. The APT (Automated Program Transformations) library [23, Topic APT] provides tools to carry out derivation steps $s_i \rightsquigarrow s_{i+1}$ via automated transformations (as outlined above) in ACL2. This paper describes the APT tools to carry out isomorphic data type refinements via transformations.

Besides deriving programs from specifications, APT is useful to verify existing code. One can build an *anti-derivation*, i.e. a sequence $p \overset{\mathbf{r}}{\mapsto} s'_0 \leftsquigarrow s'_1 \leftsquigarrow \ldots \leftsquigarrow s'_m$, where: p is an existing program; $\overset{\mathbf{r}}{\mapsto}$ yields a representation s'_0 of the program in the logical specification language; $\leftsquigarrow$ is an *anti-refinement* relation (converse of $\rightsquigarrow$); and s'_m is a more abstract representation of the program that should be easier to verify than s'_0. In the *analysis-by-synthesis* approach [11, 15], an anti-derivation $p \overset{\mathbf{r}}{\mapsto} s'_0 \leftsquigarrow s'_1 \leftsquigarrow \ldots \leftsquigarrow s'_m$ can be combined with a derivation without code generation $s_0 \rightsquigarrow s_1 \rightsquigarrow \ldots \rightsquigarrow s_n$, where s_n and s'_m are equal or trivially equivalent, to verify that p satisfies s_0. A step $s'_j \leftsquigarrow s'_{j+1}$ in an anti-derivation may be carried out (when possible) via an automated transformation that is "inverse" of one that would realize the converse step $s'_{j+1} \rightsquigarrow s'_j$ in a derivation. The isomorphic data type transformations described in this paper are readily invertible, due to the inherent symmetry of isomorphisms. (Other kinds of transformations are more difficult to invert.)

2 Mathematical Overview

Here functions are viewed not extensionally as possibly infinite sets of pairs, but intensionally as finite descriptions such as the function definitions in ACL2 and similar languages.

Consider a function $f : X \to Y$ with definition $f(x) \equiv$ **if** $a(x)$ **then** $b(x)$ **else** $c(x, f(d(x)))$, where: $a(x)$ is the termination test; $b(x)$ is the base case of the recursion; $c(x, f(d(x)))$ combines the recursive call with the argument; and $d(x)$ decreases some measure μ of the argument according to some well-founded relation $\prec$, i.e. $\neg\, a(x) \implies \mu(d(x)) \prec \mu(x)$, so that f is logically consistent. This is a representative recursive function definition, which covers special cases such as non-recursion (when $a(x)$ is always true), and whose generalization to multiple base cases and recursive calls is easily imagined. When a, b, c, and d are executable, f describes a computation from inputs in X to outputs in Y.

Consider an isomorphism $\xi : X \to X'$ with inverse $\xi^{-1} : X' \to X$ (so $\xi^{-1} \circ \xi = id_X$ and $\xi \circ \xi^{-1} = id_{X'}$), and an isomorphism $\upsilon : Y \to Y'$ with inverse $\upsilon^{-1} : Y' \to Y$ (so $\upsilon^{-1} \circ \upsilon = id_Y$ and $\upsilon \circ \upsilon^{-1} = id_{Y'}$). If ξ and υ are executable, they describe computations to change the representations of f's inputs and outputs to the ones in X' and Y'. If ξ^{-1} and υ^{-1} are also executable, they describe computations to change the representations back to the ones in X and Y.

An *isomorphic* version $f' : X' \to Y'$ of f that operates on the values in X' and Y' can be mechanically defined as $f'(x') \equiv$ **if** $a(\xi^{-1}(x'))$ **then** $\upsilon(b(\xi^{-1}(x')))$ **else** $\upsilon(c(\xi^{-1}(x'), \upsilon^{-1}(f'(\xi(d(\xi^{-1}(x')))))))$, which has the same form as f but with the four isomorphic conversions (ξ, υ, etc.) added to make it "type-correct".[1] The relationship between f' and f is $f' = \upsilon \circ f \circ \xi^{-1}$, or equivalently $f = \upsilon^{-1} \circ f' \circ \xi$,

[1] This covers the special case in which only the output data representation is changed (i.e. $X' = X$ and $\xi = \xi^{-1} = id_X$), or only the input data representation is changed (i.e. $Y' = Y$ and $\upsilon = \upsilon^{-1} = id_Y$). It also covers the case in which f has multiple

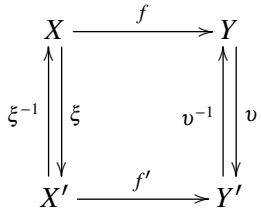

Figure 1: Relationship between f and its isomorphic version f'.

illustrated in the commuting diagram in Figure 1; this is proved by induction, using the properties of the isomorphisms. The logical consistency of f' follows from the one of f, using the measure $\mu \circ \xi^{-1}$. The computation described by f' is the same as f, with the addition of isomorphic conversions back and forth as needed.

Expanding the definitions of a, b, c, d, ξ, etc., it may be possible to use rewrite rules to simplify the body of f', obtaining an equivalent new function definition that operates directly on X' and Y' and no longer uses the isomorphic conversions and the old data representations. This is a separate process, which may require user guidance in general. But it can be automated if there exist (recursively obtained[2]) isomorphic versions a' of a with $a(x) = a'(\xi(x))$,[3] b' of b with $b(x) = \upsilon^{-1}(b'(\xi(x)))$, c' of c with $c(x,y) = \upsilon^{-1}(c'(\xi(x), \upsilon(y)))$, and d' of d with $d(x) = \xi^{-1}(d'(\xi(x))$. The latter four equalities, along with the properties of the isomorphisms, applied as rewrite rules directly in the definition of f', yield $f'(x') \equiv$ **if** $a'(x')$ **then** $b'(x')$ **else** $c'(x', f'(d'(x')))$. The automation of this process is enabled by the fact that the definition of f manipulates values in X and Y exclusively via functions (a etc.) that have isomorphic versions (a' etc.).[4] In this case, the definition of f' without the isomorphic conversions can be mechanically constructed from the definition of f, and so can the proofs of the relationship between f and f' as above. We say that the isomorphic transformations of a, b, etc. are *propagated* to f. If instead not all the functions in f that manipulate values in X and Y have isomorphic versions, we say that we *initiate* the isomorphic transformation at f. If some but not all of the functions a, b, etc. have isomorphic versions, their associated isomorphic relationships can be used in the subsequent user-guided simplification of f', but this case is still considered an initiation and not a propagation. In a well-engineered development where the data structures whose representations are being transformed are accessed via a relatively small number of interface functions, isomorphic transformations will be initiated at the interface functions and propagated to all the other functions.

In general, a function like f above may be part of (i.e. directly or indirectly called by) a requirements or intermediate specification s_i of the kind described in Section 1. For instance, a requirements or intermediate specification of a compiler may involve functions that manipulate data structures like stacks and symbol tables, which may be transformed into more efficient isomorphic data structures as part of a derivation of an implementation of the compiler. There may be several functions (f, a, etc.) manipulating these data structures. In subjecting all these functions to isomorphic transformations, some will require user-guided simplification to eliminate the isomorphic conversions, but others will be amenable

inputs (i.e. $X = X_1 \times \cdots \times X_n$ and $X' = X'_1 \times \cdots \times X'_{n'}$, with either $n = n'$ or $n \neq n'$) and/or multiple outputs (i.e. $Y = Y_1 \times \cdots \times Y_m$ and $Y' = Y'_1 \times \cdots \times Y'_{m'}$ with either $m = m'$ or $m \neq m'$).

[2]Here 'recursively' refers to the call graph of the functions.

[3]Since here a and a' return booleans, only the representation of their inputs is changed.

[4]This would be still the case if the definition of f also manipulated values in X and Y with certain functions, like equality, that operate "polymorphically" on all value representations, without the need to have isomorphic versions for these functions.

to automatic propagation, including the predicate s_i.[5]

3 Realization in ACL2

We developed three ACL2 tools to realize the isomorphic transformations described in Section 2:

1. `Defiso`, to establish isomorphic mappings like $X — \xi \succ X'$ and $Y — \upsilon \succ Y'$. See Section 3.1.
2. `Isodata`, to initiate isomorphic transformations based on `defiso` mappings. See Section 3.2.
3. `Propagate-iso`, to propagate isomorphic transformations initiated by `isodata`. See Section 3.3.

3.1 Establishing Isomorphic Mappings

Even though the ACL2 language is untyped, predicates (i.e. functions that are, or are treated as, boolean-valued) often play the role of types in ACL2. Thus, sets of data representations like X, X' etc. in Section 2 are denoted by ACL2 predicates. Isomorphisms like ξ, ξ^{-1}, etc. in Section 2 are denoted by ACL2 functions. Since functions are total in ACL2 (i.e. always defined over all values), the isomorphisms are "relativized" to the predicates, as explicated below.

`Defiso` [23, Topic `defiso`] takes four functions as (some of its) inputs:

- `old`, the predicate for the old data representation, like X or Y in Section 2.
- `new`, the predicate for the new data representation, like X' or Y' in Section 2.
- `iso`, the isomorphism from `old` to `new`, like ξ or υ in Section 2.
- `osi`, the isomorphism from `new` to `old`, like ξ^{-1} or υ^{-1} in Section 2.

`Defiso` attempts to prove the following applicability conditions (assuming unary functions):[6]

- `(old o)` $\Longrightarrow$ `(new (iso o))`, i.e. `iso` maps values in `old` to values in `new`.
- `(new n)` $\Longrightarrow$ `(old (osi n))`, i.e. `osi` maps values in `new` to values in `old`.
- `(old o)` $\Longrightarrow$ `(osi (iso o)) = o`, i.e. `osi` is `iso`'s inverse over the values in `old`.
- `(new n)` $\Longrightarrow$ `(iso (osi n)) = n`, i.e. `iso` is `osi`'s inverse over the values in `new`.

As in APT transformations, it is the user's responsibility to ensure that these conditions are proved, possibly by supplying hints to `defiso` or by proving lemmas before calling `defiso`.

If these proofs are successful, `defiso` generates theorems asserting the injectivity of `iso` and `osi` over the predicates:

- `(old o1)` $\wedge$ `(old o2)` $\Longrightarrow$ `[(iso o1) = (iso o2)` $\Longleftrightarrow$ `o1 = o2]`
- `(new n1)` $\wedge$ `(new n2)` $\Longrightarrow$ `[(osi n1) = (osi n2)` $\Longleftrightarrow$ `n1 = n2]`

The proof hints for these injectivity theorems are generated automatically from the applicability conditions [2],[7] similarly to how the APT transformations generate $\rightsquigarrow$ proofs from the applicability conditions (see Section 1).

Information about all the above functions, applicability conditions, and theorems is stored into an ACL2 table, under a name supplied as another input to `defiso`. In the call `(defiso isomap old new iso osi ...)`, `isomap` is such a name, and `...` consists of hints and other optional inputs.

[5] If X and Y are part of the required "interface" in s_0 of the program being derived, note that a wrapper $\widetilde{f} \equiv \upsilon^{-1} \circ f' \circ \xi$ can be mechanically defined along with f', provably satisfying $\widetilde{f} = f$. This is different from the definition of f' that includes the isomorphic conversions: $\widetilde{f}$ only converts the input once and the output once, while the definition of f' that includes the isomorphic conversions performs several conversions at each recursive call.

[6] Sometimes this paper uses infix notations for logical connectives and equality.

[7] If ϕ has a left inverse ϕ^{-1}, then ϕ is injective: assuming $\phi(z_1) = \phi(z_2)$, apply ϕ^{-1} to both sides to obtain $\phi^{-1}(\phi(z_1)) = \phi^{-1}(\phi(z_2))$, and use the left inverse property to conclude $z_1 = z_2$.

`Defiso` optionally, and by default, generates additional applicability conditions about the guards of the predicates and isomorphisms:

- The guards of `old` and `new` are always true, so that `old` and `new` can be applied to any value.
- The guard of `iso` includes `old`, so that `iso` can be applied to any value in `old`.
- The guard of `osi` includes `new`, so that `osi` can be applied to any value in `new`.

These ensure that certain terms involving the predicates and isomorphisms are guard-verified [2].

The predicates may take multiple arguments, whose number must suitably match the number of arguments and results of the isomorphism; two or more results are returned via `mv` [23, Topic `mv`]. Thus, the old and new data representations may be subsets of cartesian products of the universe of ACL2 values. In this case, the applicability conditions involve `mv-let` [23, Topic `mv-let`].

In addition to function symbols, untranslated lambda expressions and macro symbols can also be supplied to `defiso` to specify the predicates and isomorphisms. A macro symbol `mac` abbreviates its eta expansion `(lambda (z1 z2 ...) (mac z1 z2 ...))`, limited to just the macro's required arguments `z1`, `z2`, etc.

While `defiso` was developed to support isomorphic transformations, it may have more general uses. It automates the formulation of isomorphism theorems, and the proof of injectivity theorems from them, for predicates and functions that are not necessarily used in isomorphic transformations.

3.2 Initiating Isomorphic Transformations

Consider an isomorphic mapping `(defiso isomap old new iso osi ...)`, like $X — \xi \rightarrowtail X'$ or $Y — \upsilon \rightarrowtail Y'$ in Section 2, for changing the representation of the inputs and outputs of an ACL2 function of the form

```
(defun f (x)
  (declare (xargs :guard (g x) :measure (m x)))
  (if (a x) (b x) (c x (f (d x)))))
```

which is like f in Section 2 with the addition of a guard and an explicit measure (like μ in Section 2), assuming that $X — \xi \rightarrowtail X'$ and $Y — \upsilon \rightarrowtail Y'$ are the same (i.e. $X = Y$, $X' = Y'$, etc.).

`Isodata` [23, Topic `isodata`] takes `f` and `isomap` as (some of its) inputs, and attempts to prove the following applicability conditions:[8]

- `(old x)` $\Longrightarrow$ `(old (f x))`, i.e. `f` maps inputs with the old representation to outputs with the old representation.
- `(old x)` $\wedge$ $\neg$`(a x)` $\Longrightarrow$ `(old (d x))`, i.e. the old representation is preserved across recursive calls.
- `(g o)` $\Longrightarrow$ `(old o)`, i.e. the inputs in the guard of `f` (i.e. the ones over which `f` is "well-defined") all have the old representation. This is an optional applicability condition, present when guards are to be verified (which is the default).

If these proofs (which are the user's responsibility) succeed, `isodata` generates a new function

```
(defun f1 (x)
  (declare (xargs :guard (and (new x) (g (osi x))) :measure (m (osi x))))
  (if (mbt$ (new x))
      (if (a (osi x)) (iso (b (osi x))) (iso (c (osi x) (osi (f1 (iso (d (osi x))))))))
    nil)) ; irrelevant
```

which is like f' in Section 2, with the addition of a guard and a wrapping `if`, both explained next.

[8] The applicability conditions for `(defiso isomap ...)` can be regarded as additional, "indirect" applicability conditions of the `isodata` call.

The guard's first conjunct `(new x)` ensures that: (i) the call `(osi x)` in the second conjunct is guard-verified (see the third optional applicability condition of `defiso` in Section 3.1); and (ii) `f1` only operates (according to its guard) on inputs in the new representation, analogously to how `f` only operates on inputs in the old representation (see the third applicability condition of `isodata` above).[9] The guard's second conjunct `(g (osi x))` further ensures that `f1` only operates on the subset of the new representation isomorphic to the guard of `f`. The guard proof hints, if guards are to be verified, are generated automatically from the applicability conditions [3].

The wrapping `(if (mbt$ (new x)) ...)`, where `mbt$` [23, Topic `mbt$`] is a variant of `mbt` that requires non-`nil` instead of `t` (to avoid requiring `new` to be boolean-valued), is needed because the ACL2 language is untyped (unlike the "mathematical language" in Section 2). Without it, the termination of `f1` would not be always provable from the termination of `f` with the measure `(m (osi x))`. In this case `(mbt$ (new x))` could be negated and disjoined before `(a (osi x))`, but the form above is the most general, and the branch marked as 'irrelevant' can be anything; this branch always trivially passes guard verification, because it is unreachable under the guard. The termination proof hints are generated automatically from the applicability conditions and the termination theorem of `f` [3].

`Isodata` also generates the following theorems, whose proof hints are generated automatically from the applicability conditions [3]:

- `(old x)` $\Longrightarrow$ `(f x) = (osi (f1 (iso x)))`, which is like $f = \upsilon^{-1} \circ f' \circ \xi$ in Section 2, with the hypothesis added because the ACL2 language is untyped.
- `(new x)` $\Longrightarrow$ `(f1 x) = (iso (f (osi x)))`, which is like $f' = \upsilon \circ f \circ \xi^{-1}$ in Section 2, with the hypothesis added because the ACL2 language is untyped.
- `(new x)` $\Longrightarrow$ `(new (f1 x))`, i.e. `f1` maps inputs with the new representation to outputs with the new representation, analogously to the first applicability condition of `isodata` above.

We also generate an incompatibility theory invariant [23, Topic `theory-invariant`] preventing the first and second theorems from being simultaneously enabled.

The above is accomplished via the call `(isodata f (((x :result) isomap)) ...)`, where the doublet `((x :result) isomap)` specifies the transformation of the argument `x` and of the result via `isomap`, and `...` consists of hints and other options. The paper's supporting materials[10] includes a development of this "schematic" example.

`Isodata` supports non-recursive functions, as well as recursive functions with multiple base cases and/or recursive calls, but not yet mutually recursive functions. It supports the transformation of each argument and result with a possibly different isomorphic mapping: the second input of `isodata` is a list of doublets `((arg/res-list1 isomap1) (arg/res-list2 isomap2) ...)` that specifies that the arguments and results listed in `arg/res-listk` are transformed according to the isomorphic mapping `isomapk`; results of multi-valued functions are denoted via `:result1`, `:result2`, etc. Isomorphic mappings involving tuples of arguments or results are not yet supported. Instead of referring to existing `defisos`, it is possible to specify isomorphic mappings "inline", e.g. `(... (arg/res-list (old new iso osi)) ...)`, in which case `isodata` internally generates and uses local `defisos`.

`Isodata` also supports a `:predicate` option, `nil` by default. When set to `t`, this option specifies that the target function `old` is (treated like) a predicate, and `isodata` generates slightly different applicability conditions and results in this case. See the documentation [23, Topic `isodata`] for details.

After obtaining a function like `f1` above, the user can use APT's `simplify` transformation [7] to expand the definitions of the isomorphisms and rephrase the computations to operate directly on the

[9] The guard's second conjunct alone does not prevent an `x` outside `new` to be mapped to an `(osi x)` in `old`.

[10] File `[books]/workshops/2020/coglio-westfold/schematic-example.lisp`.

new input and output representations, without any remaining references to the isomorphisms and the old representations. This process is user-guided in general.

3.3 Propagating Isomorphic Transformations

The automated process of propagating isomorphic transformations of a set of interface functions for a data type to all the higher-level functions that use them is performed by the `propagate-iso` transformation.

The first task of `propagate-iso` is to find all the functions and theorems that reference the interface functions directly or indirectly. This is done by traversing the event history starting from the definitions of the interface functions.

Type Inference. In order to generate the isomorphism theorems relating the old functions to their isomorphic versions (as in Section 3.2) we need to know which arguments and results have the type being transformed. As ACL2 is untyped, `propagate-iso` needs to do this type inference itself. The types of arguments are found by simple traversal of the guards of the functions, and the result types are found by looking at the bodies of functions and also by looking for theorems that specify them.

Generating Dependent Isomorphisms. Frequently the data type being isomorphically transformed is a component of a larger data type such as a record, list, or map. The dependent isomorphism can be constructed by mapping each element of the larger data type using the base isomorphism if it is of the base type, otherwise using identity, which is a trivial isomorphism. `Propagate-iso` tries to construct a dependent isomorphism whenever it encounters a predicate that uses the isomorphism predicate. It uses the structure of the predicate to generate the isomorphism functions.

For example, if we have the isomorphism `(defiso isomap old new iso osi ...)` and a predicate that is a pair of `natp` and `old`

```
(defun P (x) (and (natp (car x)) (old (cdr x))))
```

we get the isomorphic predicate and conversion functions

```
(defun P-new (x) (and (natp (car x)) (new (cdr x))))
(defun P-to-P-new (x) (cons (car x) (iso (cdr x))))
(defun P-new-to-P (x) (cons (car x) (osi (cdr x))))
```

An example of the dependent isomorphism for a list of `old` is given in Section 4.3.

Generated Theorems and Proof Hints. All of the functions and theorems produced by `propagate-iso` must be accepted by ACL2. This means the guard and termination conditions as well as the theorems must be proved by ACL2. We can give a meta-level argument that all these must be true, but ACL2 has to prove them at the object level. To generate hints for the ACL2 proofs, `propagate-iso` maintains three rulesets [23, Topic `rulesets`]: one for old-to-new rules, one for new-to-old rules, and one for generally-useful rules, mainly typing rules and the `defiso` theorems, with a few list data-structure rules.

For the transformed definitions and theorems, transformed proofs can be performed in principle. ACL2 does not retain proofs so this is not possible, but it can be approximated by use of transformed hints. This works when the hints are explicit, but does not take account of the proving environment, in particular, whether rules have been enabled or disabled externally since introduction.

An alternative to reproducing the original proof is to prove the new theorem from the old theorem. To do this `propagate-iso` generates a hint with a `:use` of the old theorem instantiated with the necessary new-to-old conversions, and a theory consisting of the general ruleset and the new-to-old ruleset.

Because these proof hint strategies are not guaranteed to succeed, `propagate-iso` tries both approaches: it gives instructions to first try to prove using the transformed hints and if that does not succeed, try to prove using the old theorem. As a last resort it tries proving with the current theory.

To prove the isomorphism theorems for a new introduced function, first the new-to-old theorem is proved. This is necessary in general because all the theorems on the old function are available, but the corresponding theorems on the new function will not be introduced until after the two isomorphism theorems. The rules for mapping previously-defined new functions into old functions have few preconditions so occurrences of new functions are transformed into expressions involving old functions, and then the old theorems can be used to complete the proofs. The old-to-new theorem follows easily from the new-to-old theorem and the isomorphism theorems.

As the proof hints generated by `propagate-iso` are not alway guaranteed to succeed, syntax is provided to allow the user to augment or override the generated hints for any theorem. At present, none of our examples require this level of user intervention.

4 Examples

4.1 Unbounded and Bounded Integers

Popular programming languages like C and Java typically use bounded integer types and operations, while requirements specifications typically use unbounded integer types and operations. Thus, synthesizing a C or Java program from a specification, or proving that a C or Java program complies with a specification, often involves showing that unbounded and bounded integers are "equivalent" under the preconditions stated by the specification.

When using APT as outlined in Section 1 for these synthesis or verification tasks, showing that equivalence amounts to transforming between unbounded and bounded types and operations. Previous work [7] exemplified a general methodology to transform bounded integer operations into unbounded integer operations via rewriting; it also alluded at isomorphic data type transformations for transforming bounded integer types into unbounded integer types. An unbounded type is not isomorphic to a bounded type for cardinality reasons, but a finite subset of the former may be isomorphic to the latter. When the representation of the bounded type is not just a range of integers (see examples below), there is a non-trivial isomorphism between the bounded type and the corresponding subset of the unbounded type.

The aforementioned previous work [7] uses, as an example, this Java implementation of Bresenham's line drawing algorithm (whose details are completely unimportant here):

```
// draw a line from (0, 0) to (a, b), for a and b below a given limit:
static void drawLine(int a, int b) {
    int x = 0, y = 0, d = 2 * b - a;
    while (x <= a) {
        drawPoint(x, y); // on screen
        x++;
        if (d >= 0) { y++; d += 2 * (b - a); }
        else { d += 2 * b; }
    }
}
```

The ACL2 representation of the Java code (see the cited paper for details) has the form

```
(defun drawline-loop (a b x y d screen) ; loop of the method
  (if (invariant a b x y d)
      (if (not (lte32 x a))
          screen ; exit loop
        (drawline-loop a b
                       (add32 x (int32 1))
```

```
                      (if (gte32 d (int32 0))
                          (add32 y (int32 1))
                        y)
                      (if (gte32 d (int32 0))
                          (add32 d (mul32 (int32 2) (sub32 b a)))
                        (add32 d (mul32 (int32 2) b)))
                      (drawpoint x y screen)))
    :undefined))
(defun drawline (a b screen) ; method
  (if (precondition a b)
      (drawline-loop a b
                     (int32 0) ; x
                     (int32 0) ; y
                     (sub32 (mul32 (int32 2) b) a) ; d
                     screen)
    :undefined))
```

where: add32, sub32, mul32, lte32, and gte32 are 32-bit signed integer operations; and int32 converts from (lambda (x) (signed-byte-p 32 x)) to a type of unbounded integers recognized by int32p. For the purpose of this example, the exact definitions of int32p, int32, add32, etc. are unimportant—the paper's supporting materials[11] introduce them as constrained functions. For concreteness, they could be int-value-p [23, Topic int-value-p], make-int-value [23, Topic make-int-value], int-add [23, Topic int-add], etc. in the formalization of Java primitive values and operations in [books]/kestrel/java/language; or they could be (lambda (x) (unsigned-byte-p 32 x)), (lambda (x) (loghead 32 x)), (lambda (x y) (bvplus 32 x y)), etc. in the bit vector library in [books]/kestrel/bv.

Consider the isomorphic mapping

```
(defiso isomap ; name
        int32p ; old representation
        (lambda (x) (signed-byte-p 32 x)) ; new representation
        int ; conversion from old to new representation
        int32) ; conversion from new to old representation
```

where int converts from int32p to (lambda (x) (signed-byte-p 32 x)); again, the exact definition of int is unimportant, but for concreteness it could be int-value->int [23, Topic int-value] in the formalization of Java primitive values and operations in [books]/kestrel/java/language, or (lambda (x) (logext 32 x)) in the bit vector library in [books]/kestrel/bv.

Applying isodata to drawline-loop and drawline with isomap, changes the representation of a, b, x, y, and d from int32p to (lambda (x) (signed-byte-p 32 x)). This puts int32 in front of each occurrence of a, b, etc., and puts int in front of each recursive call argument, which produces some terms of the form (int (int32 ...)):

```
(defun drawline-loop1 (a b x y d screen)
  ...
  (if (not (lte32 (int32 x) (int32 a)))
      screen
    (drawline-loop1 (int (int32 a))
                    (int (int32 b))
                    (int (add32 (int32 x) (int32 1)))
                    (int (if (gte32 (int32 d) (int32 0))
                             (add32 (int32 y) (int32 1))
```

[11] File [books]/workshops/2020/coglio-westfold/integer-example.lisp.

```
                        (int32 y)))
                  (int (if (gte32 (int32 d) (int32 0))
                           (add32 (int32 d)
                                  (mul32 (int32 2)
                                         (sub32 (int32 b) (int32 a))))
                           (add32 (int32 d)
                                  (mul32 (int32 2) (int32 b)))))
                  (drawpoint (int32 x) (int32 y) screen)))
  ...)
(defun drawline1 (a b screen)
  ...
  (drawline-loop1 (int32 a) (int32 b) (int32 0) (int32 0)
                  (sub32 (mul32 (int32 2) (int32 b))
                         (int32 a))
                  screen)
  ...)
```

Then applying APT's `simplify` [7] with the rewrite rules for transforming `add32` to `+` etc., which temporarily produces further terms of the form `(int (int32 ...))`, results in the following functions, whose bodies no longer use the bounded type and operations (except for the call in `drawpoint`, but that could be also similarly transformed):

```
(defun drawline-loop2 (a b x y d screen)
  ...
  (if (< a x)
      screen
    (drawline-loop2 a b
                    (+ 1 x)
                    (if (< d 0) y (+ 1 y))
                    (if (< d 0)
                        (+ d (* 2 b))
                      (+ d (- (* 2 a)) (* 2 b)))
                    (drawpoint (int32 x) (int32 y) screen)))
  ...)
(defun drawline2 (a b screen)
  ...
  (drawline-loop2 a b 0 0 (+ (- a) (* 2 b)) screen)
  ...)
```

The technique exemplified above extends the technique exemplified in previous work [7], from transforming just bounded integer operations into unbounded ones, to transforming both bounded integer types and bound integer operations into unbounded ones.

4.2 Loop Re-Indexing

A perhaps less expected example of isomorphic data type transformation is the re-indexing of a loop to count up instead of down (or vice versa). In this case, the old and new representation are the same, but the isomorphic conversions "flip" the loop range.

As a simple but suggestive example, consider the tenfold application of a unary function `h`:[12]

```
(defun applyn (x n) ; apply h to x for n times -- (h (h (h ... (h x)...)))
  (declare (xargs :guard (and (natp n) (<= n 10))))
```

[12] Fixed-length loops are common in cryptography, for instance.

```
    (if (zp n) x (h (applyn x (1- n)))))
  (defun applyten (x) ; apply h to x for 10 times
    (declare (xargs :guard t))
    (applyn x 10))
```

The two functions above could be part of a requirements or intermediate specification. Presumably a requirements specification would omit the condition `(<= n 10)` from `applyn`, but that condition could be added by the `restrict` transformation [23, Topic `restrict`] as a preparatory refinement step prior to the ones described below, given that `applyten` calls `applyn` with 10 as `n`: this way, `applyn` above would be part of an intermediate specification. Regardless, `applyn` exhibits a simple recursion on the natural numbers with measure and termination proof easily found by ACL2. This "loop" counts down, from 10 to 0, which is natural in a functional program or specification.

The set $\{0,\ldots,10\}$ of the possible values of the loop variable `n` is isomorphic to itself in more than one way (i.e. the obvious way, where the isomorphisms are identity). In particular, the function that maps each element $n \in \{0,\ldots,10\}$ to $10-n$ is an isomorphism over the set, with itself as the inverse. The following isomorphic transformation step, where the `defiso` is generated on the fly, succeeds:

```
(isodata applyn ((n ((lambda (n) (and (natp n) (<= n 10)))
                     (lambda (n) (and (natp n) (<= n 10)))
                     (lambda (n) (- 10 n))
                     (lambda (n) (- 10 n)))))
        :new-name applyn0)
```

The resulting function includes arithmetic expressions such as `(+ 10 (- (+ -1 10 (- n))))`, amenable to simplification via APT's `simplify` transformation [7], which produces

```
(defun applyn1 (x n)
  (declare (xargs :guard ... :measure (acl2-count (+ 10 (- n)))))
  (if (and (natp n) (<= n 10))
      (if (< n 10)
          (h (applyn1 x (+ 1 n)))
        x)
    nil))
```

Now the loop counts up, from 0 to 10, which is more complicated in a functional program or specification (see the measure of `applyn1`), but more natural or common in an imperative implementation, which is the direction where this loop re-indexing transformation is headed.

See the paper's supporting materials[13] for a complete development of this example.

For this simple example, one could manually write the new functions, and the theorems that relate them to the old functions, without any additional proof effort: ACL2 proves the theorems automatically. However, calling `isodata` and `simplify` seems easier, and explicates the principles that lead from the old loop to the new loop. With more complex examples, the manually written theorems may no longer be proved automatically, perhaps due to "interference" from other rules that apply to other parts of the functions. In contrast, `isodata` is designed to generate proof hints for termination, guards, and theorems that are precisely targeted to the proof goals (e.g. that do not depend on the current theory) and that are expected to always succeed. With reference to Section 1, manually transforming a loop amounts to writing s_{i+1} and verifying $s_i \rightsquigarrow s_{i+1}$, while using automated transformations amounts to generating s_{i+1} and the proof of $s_i \rightsquigarrow s_{i+1}$ from s_i.

[13] File `[books]/workshops/2020/coglio-westfold/loop-example.lisp`.

4.3 Efficient Value Caching With Invariant Maintenance

Our largest example is a drone route planner. The problem is to have a set of drones visit a set of sites. This is a distributed system with route planning interleaved with execution. At each step each drone generates candidate plans for where to visit next. These plans are sent to a coordinator that filters these candidate plans to reduce redundancy among the different drones. Each drone then chooses one of the filtered plans to execute. After execution of the (partial) plan, the plan-coordinate-execute cycle is repeated until all the target locations have been visited.

The key data type is the state of a drone, `drone-st`. A list of drone states, one for each drone, is passed around the high-level functions. The drone state has four fields: the identifier for the drone; a graph which gives the possible moves from each location (`node`); the nodes that have been visited; and the path that has been taken so far in the planning process:[14]

```
(fty::defprod dr-state
  ((drone-id drone-id)
   (dgraph dgraph-p)
   (visited-nodes node-list)
   (path-taken node-list)))
```

During the planning, it is important to know the current location of each node and the nodes that are unvisited at each point in the planning process. A natural optimization is to store these values in the drone state record and maintain them incrementally instead of recomputing them from the path taken by the drone and its set of visited nodes.

Isomorphism Definition. The first step of this optimization is to define a new state record with fields for the values to be maintained:

```
(fty::defprod dr-state-ext0
  ((drone-id drone-id)
   (dgraph dgraph-p)
   (visited-nodes node-list)
   (path-taken node-list)
   (unvisited-nodes node-list)
   (currentpos node-or-null)))
```

We define functions to map back and forth between these two types: `from-dr-state-ext`, which just omits the two new fields; and `to-dr-state-ext` which computes the values of the two new fields. We also define theorems to relate the accessor functions of the two types using these functions. For example:

```
(defthm dr-state->drone-id-~>dr-state-ext0->drone-id
  (equal (dr-state->drone-id drn-st)
         (dr-state-ext0->drone-id (to-dr-state-ext drn-st))))
```

To define a predicate representing a type isomorphic to `dr-state`, we restrict the two new fields to be functions of the other fields:

```
(define dr-state-ext-p (drn-st)
  :returns (b booleanp)
  (and (dr-state-ext0-p drn-st)
       (equal (dr-state-ext0->unvisited-nodes drn-st)
              (set-difference-equal (dgraph->nodes (dr-state-ext0->dgraph drn-st))
                                    (dr-state-ext0->visited-nodes drn-st)))
       (equal (dr-state-ext0->currentpos drn-st)
              (drone-location (from-dr-state-ext drn-st)))))
```

[14] The use of the FTY [23, Topic `fty`] package here is not essential. Any product/record constructs could be used.

The isomorphism is then:

```
(defiso dr-state-p-to-dr-state-ext-p
  dr-state-p dr-state-ext-p to-dr-state-ext from-dr-state-ext
  :hints (:beta-of-alpha (("Goal" :in-theory (enable from-dr-state-ext to-dr-state-ext)))
          :alpha-of-beta (("Goal" :in-theory (enable from-dr-state-ext to-dr-state-ext)))))
```

Given the basic isomorphism we are now ready to perform the isomorphic type refinement. Apart from the accessor functions, `extend-path-taken` is the only function that accesses the internals of `dr-state`. It is the function that incorporates a new plan into the state, extending the `path-taken` and adding the plan nodes to the `visited-nodes`.

```
(define extend-path-taken ((drn-st good-dr-state-p) (plan node-list-p))
  :guard (or (null plan) (valid-plan-p (drone-location drn-st) plan drn-st))
  (change-dr-state drn-st
                   :path-taken (append (dr-state->path-taken drn-st) plan)
                   :visited-nodes (union-equal (dr-state->visited-nodes drn-st)
                                               plan)))
```

Initial Propagation. The guard uses three functions that depend on `dr-state`: `good-dr-state-p`, `valid-plan-p` and `drone-location`, where `good-dr-state-p` and `valid-plan-p` are simple well-formedness predicates on `dr-state` and `plan` types respectively. We want isomorphic versions of these three functions before transforming `extend-path-taken`. As these functions do not access `dr-state` directly, their isomorphic versions can be generated using `propagate-iso`:

```
(propagate-iso dr-state-p-to-dr-state-ext-p
               ((dr-state dr-state-ext  ; Constructors
                 dr-state-~>dr-state-ext dr-state-ext-~>dr-state
                 (* * * *) => (dr-state-p))
                (dr-state->drone-id dr-state-ext0->drone-id   ; Destructors
                 dr-state->drone-id-~>dr-state-ext0->drone-id
                 dr-state-ext0->drone-id-~>dr-state->drone-id
                 (dr-state-p) => *)
                 ...
               --Similar entries for the other 3 destructors--
               )
               :first-event dr-state*-p
               :last-event valid-plan-p-node-path-p)
```

The first argument `dr-state-p-to-dr-state-ext-p` is the name of the isomorphism, which could in general be a list of isomorphisms to be applied in parallel. Then follows information about the interface functions, in this case just the constructor and destructor functions. For example, the first element specifies that for the constructor function `dr-state`, `dr-state-ext` is the isomorphic constructor function, `dr-state-~>dr-state-ext` and `dr-state-ext-~>dr-state` are the theorems that can transform one into the other, and `(* * * *) => (dr-state-p)` is the signature of `dr-state`. This signature specifies that `dr-state` takes four arguments not of any isomorphism type and returns a single value satisfying the `dr-state-p` predicate. The `:first-event` and `:last-event` specify the range of events for which to consider creating isomorphic versions.

Dependent Isomorphisms. For predicates that include a call to an existing isomorphism predicate, `propagate-iso` not only creates an isomorphic version of the predicate, but also constructs an isomorphism between the two. For example, for the predicate `all-dr-state-p` defined as

```
(defun all-dr-state-p (drn-sts)
  (if (atom drn-sts)
      (null drn-sts)
    (and (dr-state-p (first drn-sts))
         (all-dr-state-p (rest drn-sts)))))
```

it defines the isomorphic predicate `all-dr-state-ext-p` as

```
(defun all-dr-state-ext-p (drn-sts)
  (if (atom drn-sts)
      (null drn-sts)
    (and (dr-state-ext-p (first drn-sts))
         (all-dr-state-ext-p (rest drn-sts)))))
```

and, based on the structure of the predicate, creates the conversion functions

```
(defun all-dr-state-p-->-all-dr-state-ext-p (drn-sts)
  (if (atom drn-sts)
      nil
    (cons (to-dr-state-ext (first drn-sts))
          (all-dr-state-p-->-all-dr-state-ext-p (rest drn-sts)))))
(defun all-dr-state-ext-p-->-all-dr-state-p (drn-sts)
  (if (atom drn-sts)
      nil
    (cons (from-dr-state-ext (first drn-sts))
          (all-dr-state-ext-p-->-all-dr-state-p (rest drn-sts)))))
```

and creates the isomorphism

```
(defiso all-dr-state-p-iso-all-dr-state-ext-p
  all-dr-state-p all-dr-state-ext-p
  all-dr-state-p-->-all-dr-state-ext-p all-dr-state-ext-p-->-all-dr-state-p
  :hints ...)
```

Last Initialization. After `propagate-iso` has generated isomorphic versions of the three functions used in its guard, the isomorphic version of `extend-path-taken` is generated using `isodata` and optimized by `simplify`:

```
(isodata extend-path-taken (((drn-st :result) good-dr-state-p-iso-good-dr-state-ext-p)))
```

which specifies that the `drn-st` argument and the result are of the isomorphism source type, and generates the isomorphic version of `extend-path-taken`:[15]

```
(define extend-path-taken$1 ((drn-st good-dr-state-ext-p) (plan node-list-p))
  (to-dr-state-ext (let ((new-drn-st (from-dr-state-ext drn-st)))
                     (dr-state (dr-state->drone-id new-drn-st)
                               (dr-state->dgraph new-drn-st)
                               (union-equal (dr-state->visited-nodes new-drn-st)
                                            plan)
                               (append (dr-state->path-taken new-drn-st)
                                       plan)))))
```

Then the `simplify` form:

```
(simplify extend-path-taken$1
          :assumptions ((good-dr-state-ext-p drn-st))
          :enable (dr-state-ext set-difference-equal-2-append-rev))
```

produces the optimized version:

[15] Currently `isodata` and `simplify` produce results that are equivalent to these but a little more syntactically complex, using `defun`, `mbt` and extra let-binding. The simplified form presented here is intended to facilitate comparison with the original definition of `extend-path-taken`. Also, the guards are omitted for brevity.

```
(define extend-path-taken$2 ((drn-st good-dr-state-ext-p) (plan node-list-p))
  (dr-state-ext0 (dr-state-ext0->drone-id drn-st)
                 (dr-state-ext0->dgraph drn-st)
                 (union-equal (dr-state-ext0->visited-nodes drn-st) plan)
                 (append (dr-state-ext0->path-taken drn-st) plan)
                 (set-difference-equal (dr-state-ext0->unvisited-nodes drn-st)
                                       plan)
                 (if (consp plan)
                     (car (last plan))
                   (dr-state-ext0->currentpos drn-st))))
```

`Simplify` not only eliminates the conversion functions `to-dr-state-ext` and `from-dr-state-ext`, but the enabling of the interface function `dr-state-ext` and specific distributive rules allow the `unvisited-nodes` and `currentpos` fields to be computed incrementally rather than using the relatively expensive expressions in the definition of `dr-state-ext-p` above.

`Isodata` produces theorems relating `extend-path-taken` and `extend-path-taken$1`; `simplify` produces theorems relating `extend-path-taken$1` and `extend-path-taken$2`. However `propagate-iso` requires theorems relating `extend-path-taken` and `extend-path-taken$2`, which are proved by simply chaining the `isodata` and `simplify` theorems.

Final Propagation. The simplest way to call `propagate-iso` to complete the isomorphism propagation is with the same arguments as before, but just adding isomorphism information about `extend-path-taken` and changing the `:last-event`. This regenerates the events from the previous call, but ACL2's redundancy checking makes this cheap. An alternative is to also change the `:first-event`, but then the tables from the previous call must be given to the new call. There is an option to print these out in the right form for the following call, but they can be large so it is much more concise and also less brittle to have `propagate-iso` regenerate them.

For this drone planner, `propagate-iso` generates 32 translated functions, 6 isomorphisms including defining 4 to-and-from translator functions, and 317 non-local theorems, all of which are proven with the automatically generated hints.

5 Related Work

The idea of isomorphic data type transformations is not new. In particular, the authors of this paper worked on the design and implementation of a similar transformation for the Specware system [16]. The novel contributions of this paper are: (i) the mathematical characterization of this transformation with the 'initiation' and 'propagation' nomenclature in Section 2; and (ii) the realization in the ACL2 theorem prover described in Section 3 and exemplified in Section 4. Since the Specware language is typed and higher-order, the Specware version of the transformation differs from the APT/ACL2 version in some important aspects. More broadly, the Specware transformation system, which drew inspiration from the KIDS transformation system [19], is the closest work to APT,[16] and in fact provided much inspiration to it. A difference between Specware and APT is that the latter is tightly integrated with a theorem prover (ACL2), while the former had interfaces and translations to external theorem provers.

Automated type transformation tools exists for typed higher-order theorem provers like Isabelle/HOL and Coq [17, 18, 9, 8]. These tools support richer mappings than isomorphisms, but it is not immediately apparent how they fit the paradigm in Section 2 of taking as inputs any[17] function f and isomorphic

[16] The generic 'APT' refers to the whole library of transformation tools, not just the isomorphic type transformation ones.

[17] Provided that the applicability conditions hold, obviously.

mappings and returning as outputs a function f' and theorems relating f and f'. In particular, some of the referenced tools are built for infrastructures consisting of refinement monads and libraries of verified data types and relative operations. The referenced tools and APT's isomorphic type transformation tools may mutually benefit from incorporating some of each other's ideas, but many technical aspects are bound to differ due to the inherent differences between ACL2 and typed higher-order theorem provers. More broadly, while the aforementioned refinement monads follow a well-established refinement approach based on inclusion of sets of behaviors, APT is better suited to the pop-refinement approach, mentioned in Section 1, based on inclusion of sets of implementations, which in particular provides more flexibility and clarity in matters of non-determinism, under-specification, and hyperproperties [4, Section 4.4].

Smith [20] developed the concept of connections between theories, which generates similar definitions for datatype functions, but based on homomorphisms between types rather than isomorphisms. Connections arose by generalizing from transformations implemented in the KIDS system [19].

6 Future Work

`Defiso` may already provide all the features needed for isomorphic transformations. Any future extensions may be driven by non-APT uses of this more general tool.

`Isodata` should be extended to overcome the limitations noted in Section 3.2. Eventually, `isodata` should be able to partition all the n arguments and m results of a function into disjoint subsets, and apply a different isomorphic mapping to each subset (possibly the identity one, for arguments and results to leave unchanged); each such isomorphic mapping may map tuples (i.e. multiple arguments or results) to tuples of possibly different sizes. This will enable the use of `isodata` for perhaps less expected transformations, such as reordering arguments/results, renaming arguments, grouping multiple arguments/results into single list arguments/results, ungrouping single list arguments/results into multiple arguments/results, and adding redundant arguments (caching) for incrementalizing computations. The realization that so many apparently different kinds of transformations can be unified under the `isodata` umbrella, contributes to the quest for a "minimal" and "complete" set of program transformations; it is an open question whether such a set exists.

`Propagate-iso` needs to be tested on more kinds of examples. There is likely incremental improvement possible for the hints it produces. This is difficult in general because most of the theorems produced need to be used as well as proved: adding hypotheses can make the proof easier, but it makes them more difficult to use. Also, we would like to generalize the automatic creation of dependent isomorphisms, which currently handle predicates using list destructors. It should be straightforward to additonally handle predicates using destructors such as those from `defaggregate`. Calling `propagate-iso` could be made easier for the user if more information were automatically extracted from theorems in the world.

While isomorphisms cover a wide variety of mappings, many mappings of interest are not isomorphisms. We have worked out a preliminary design for a transformation that is similar to `isodata` but allows richer mappings. Some of its underlying concepts are similar to `isodata`, but there are necessarily some new concepts as well. In the aforementioned quest for a minimal and complete set of transformations, `isodata` may turn out to be a special case of this upcoming new transformation.

References

[1] Jean-Raymond Abrial (1996): *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, doi:10.1017/CBO9780511624162.

[2] Alessandro Coglio: *Design Notes for* `defiso`. `http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/res/kestrel-std-util-design-notes/defiso.pdf`.

[3] Alessandro Coglio: *Design Notes for* `isodata`. `http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/res/kestrel-apt-design-notes/isodata.pdf`.

[4] Alessandro Coglio (2014): *Pop-Refinement*. *Archive of Formal Proofs*. `http://afp.sf.net/entries/Pop_Refinement.shtml`, Formal proof development.

[5] Alessandro Coglio (2015): *Second-Order Functions and Theorems in ACL2*. In: *Proc. 13th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2015)*, Electronic Proceedings in Theoretical Computer Science (EPTCS), pp. 17–33, doi:10.4204/EPTCS.192.3.

[6] Alessandro Coglio (2018): *A Simple Java Code Generator for ACL2 Based on a Deep Embedding of ACL2 in Java*. In: *Proc. 15th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2018)*, Electronic Proceedings in Theoretical Computer Science (EPTCS), pp. 1–17, doi:10.4204/EPTCS.280.1.

[7] Alessandro Coglio, Matt Kaufmann & Eric Smith (2017): *A Versatile, Sound Tool for Simplifying Definitions*. In: *Proc. 14th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2017)*, pp. 61–77, doi:10.4204/EPTCS.249.5.

[8] Cyrill Cohen, Maxime Dénès & Anders Mörtberg (2013): *Refinements for Free!* In: *Proc. 3d International Conference on Certified Programs and Proofs (CPP)*, *LNCS* 8307, Springer, pp. 147–162, doi:10.1007/978-3-319-03545-1_10.

[9] Benjamin Delaware, Clement Pit-Claudel, Jason Gross & Adam Chlipala (2015): *Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant*. In: *Proc. 42nd ACM Symposium on Principles of Programming Languages (POPL)*, doi:10.1145/2676726.2677006.

[10] Edsger W. Dijkstra (1968): *A Constructive Approach to the Problem of Program Correctness*. *BIT* 8(3), doi:10.1007/BF01933419.

[11] Cordell Green (2014): *The 'Analysis-By-Synthesis' Idea*. Private Communication.

[12] C. A. R. Hoare (1972): *Proof of Correctness of Data Representations*. *Acta Informatica* 1(4), pp. 271–281, doi:10.1007/BF00289507.

[13] Cliff Jones (1990): *Systematic Software Development using VDM*, second edition. Prentice Hall.

[14] Kestrel Institute: *APT*. `https://www.kestrel.edu/home/projects/apt`.

[15] Kestrel Institute: *DerivationMiner*. `https://www.kestrel.edu/home/projects/derivationminer`.

[16] Kestrel Institute: *Specware*. `http://www.specware.org`.

[17] Peter Lammich (2019): *Refinement to Imperative HOL*. *Journal of Automated Reasoning* 62(4), pp. 481–503, doi:10.1007/s10817-017-9437-1.

[18] Andreas Lochbihler (2013): *Light-weight containers for Isabelle: Efficient, Extensible, Nestable*. In: *Proc. 4th International Conference on Interactive Theorem Proving (ITP)*, *LNCS* 7998, Springer, pp. 116–132, doi:10.1007/978-3-642-39634-2_11.

[19] Douglas R. Smith (1990): *KIDS: A Semi-Automatic Program Development System*. *IEEE Transactions on Software Engineering — Special Issue on Formal Methods* 16(9), pp. 1024–1043, doi:10.1109/32.58788.

[20] Douglas R. Smith (1993): *Constructing Specification Morphisms*. *Journal of Symbolic Computation, Special Issue on Automatic Programming* 15(5–6), pp. 571–606, doi:10.1016/S0747-7171(06)80006-4.

[21] Douglas R. Smith (1999): *Mechanizing the Development of Software*. In Manfred Broy, editor: *Calculational System Design, Proc. Marktoberdorf Summer School*, IOS Press.

[22] J. M. Spivey (1992): *The Z Notation: A Reference Manual*, second edition. Prentice Hall.

[23] The ACL2 Community: *The ACL2 Theorem Prover and Community Books: Documentation*. `http://www.cs.utexas.edu/~moore/acl2/manuals/current/manual`.

[24] Niklaus Wirth (1971): *Program Development by Stepwise Refinement*. *Communications of the ACM* 14(4), pp. 221–227, doi:10.1145/362575.362577.