

# INTEGRATING RUNTIME ASSERTIONS WITH DYNAMIC TYPES: STRUCTURING DERIVATION FROM AN INCOMPUTABLE SPECIFICATION

Paul A. Bailes

Colin J.M Kemp

School of Information Technology and Electrical Engineering

The University of Queensland QLD 4072

AUSTRALIA

{paul, ck}@itee.uq.edu.au

## ABSTRACT

An inherent incomputability in the specification of a functional language extension that combines assertions with dynamic type checking is isolated in an explicit derivation from mathematical specifications. The combination of types and assertions (into “dynamic assertion-types” – DATs) is a significant issue since, because the two are congruent means for program correctness, benefit arises from their better integration in contrast to the harm resulting from their unnecessary separation. However, projecting the “set membership” view of assertion-checking into dynamic types results in some incomputable combinations. Refinement of the specification of DAT checking into an implementation by rigorous application of mathematical identities becomes feasible through the addition of a “best-approximate” pseudo-equality that isolates the incomputable component of the specification. This formal treatment leads to an improved, more maintainable outcome with further development potential.

## KEY WORDS

Functional Programming; Language Design; Assertions; Type-checking

## 1. CORRECTNESS, TYPES AND ASSERTIONS

Type- and assertion-systems are valuable means of achieving program correctness, and should therefore themselves be developed rigorously with their own correctness in mind. In this development however arises an interesting problem, i.e. the refinement of non-computable specifications, our approach to the solution of which may identify a useful general approach to such problems.

### 1.1. Need for Types

The problem addressed by type-checking is that “within” a program there exist different types of object

each of whose behaviours is characterised algebraically, i.e. in terms of specific operators thereon. Mis-application to an object of an operation outside the algebra is highly-indicative of an error. In order to detect such domain incompatibilities [1], it's essential either that the underlying machine/interpreter maintain distinct representations for different types so that mis-applications can be detected dynamically, or that mis-applications be detected by static analysis (so that a "typeless" representation can be employed, e.g. machine code as compiler output). The former is what we know as "dynamic", the latter as "static" typing. In order to guard most strongly against domain incompatibilities, some degree of dynamic type-checking is at least oftentimes appropriate (e.g. for implementation of overloading or subclassing [2]), if not always available.

## 1.2. Need for Assertions

Assertions have a long pedigree [3] in formal specification, verification and derivation of software. A complementary role for assertions in implementations is their retention in source code in order to check that execution of an implementation accords with its specification (e.g. as a "failsafe" against derivation errors). Following the specification, an implementation may typically retain "precondition" assertions between a component's inputs and "postcondition" assertions between its inputs and outputs (or others, such as loop invariants). As well as being the logical basis for a spectrum of manual to automatic program synthesis methods, assertions may be used operationally to check the correctness of an implementation during execution. Because assertions are essentially predicates over a (component of a) program's state, an assertion  $A$  can simply be implemented as some such "**if not  $A$  then abort**" where "abort" practically may more elegantly take the form of raising an exception etc. Note that in this operational mode, the implementation does not guarantee each assertion is *necessarily* satisfied; rather, failure of an assertion to raise an exception merely indicates that all executions of the assertion (so far) have been satisfactory.

## 1.3. Correct Derivations of Type- and Assertion-Systems

It's clearly appropriate that systems of type and assertion definition and checking be developed with the same regard to correctness as these systems aim to support. In particular, formal derivation of implementations from specifications should be undertaken. However, under the broadened view of types and assertions which results from their necessary and beneficial combination (see below), it emerges that some necessary parts of the specification are not computable. How then to handle these incomputabilities is the essential purpose of this paper (with potential applications beyond the present application about dynamic types and assertions).

At some stages in the derivation of an implementation, it will be necessary for the purpose of achieving computability to take steps that are not correctness-preserving. We are primarily interested in the regulation of such steps. One solution might be separately and carefully to justify and document each such step, but which we reject because of its unreliability (not to say inelegance) in that non-correctness-preserving transformations are by their nature unreliable, and the more we posit the less reliable does their combination become. The solution we advocate instead is to isolate a minimal necessary basis of pseudo-identities, which can be explained and comprehended in isolation, and then used judiciously in the derivations to achieve a transformation into a computable implementation. The remainder of this paper demonstrates this technique in the context of the integration of dynamic types and assertions.

## **2. TYPES AND ASSERTIONS NEED TO BE INTEGRATED**

The congruency between types and assertions strongly indicates that they would benefit from integration.

### **2.1. Compatibility of Assertions and Types**

The shared essence of types and assertions cries out for their combination. We saw above that the use of assertions for run-time correctness checking involves the application of a predicate to some component of a program's state, which if satisfied allows the computation to proceed. It's possible to view types, either static or dynamically-checked, in much the same way as follows.

First, the difference between static and dynamic types is one of degree rather than of substance, just as is the case with assertions. Static typing can be thought of as an attempt to prove that domain constraints are satisfied for every possible execution of a program, but if the constraints are too complex, then they are implemented by dynamic typing, which at least implements the constraints for each individual execution. By comparison, the logical interpretation of assertions (pre- & post-conditions, invariants etc.) as specifications whose satisfaction may be attempted to be guaranteed by a program synthesizer or verifier is similarly supplemented by runtime checking under the operational interpretation.

Second, failure of a type-check results in abortion of the computation, just as is the case with assertions. In the case of static type-checks, the abortion is somewhat masked by the facts that the computation never actually begins and that the compiler/interpreter will typically continue to check for type errors rather than abort itself, but semantically the results are identical, i.e. undefined or " $\perp$ ".

At this stage, it would seem obvious that static types are best thought of as special cases of the logical

interpretation of assertions – domain constraints that can be guaranteed by analysis of source code. A remaining apparent difference, that type-constraints apply to individual variables/parameters whereas assertions are more wide-ranging, is not so sustainable when we consider various aggregate types in programming languages (records, etc.).

## **2.2. Uniting Assertions and Types**

It's therefore remarkable that the incarnations of types and assertions in actual programming languages have such dissimilar presentations and consequently actual capabilities. As well as representing a lack of facility, this disunity adds the complexity of a programming language and only serves to distract and detract from the cause of correctness and overall quality. Accordingly, our focal problem is introduce a means which shows promise for the engineering of a scheme of dynamic assertions that incorporates the capability of types; in particular, that as well as combining assertions by means of set-theoretic/logical operators, type-theoretic combinations of assertions should also be supported.

Note that this is not to say that the combination we achieve is necessarily ideal or even complete, but rather that the problem is at least sufficiently-demanding to the extent that even the germ of a good solution has in turn a derivation that is sufficiently-complex to illustrate our approach to the refinement of incomputable specifications.

## **3. INTRODUCING “DATS”**

Our integration of dynamic types and assertions – so-called “Dynamic Assertion-Types” (DATs) – exploits advanced functional programming concepts. (See Appendix B for summary of the implementation as eventually derived.)

### **3.1. Expressively-Complete Host Language**

In order to facilitate formal derivation, it's important that the DATs be implemented at source-code rather than implementation (compiler/interpreter) level. In other words, DAT constructs should be directly-expressible in the language for which the DAT facility is being derived. In order to maximize the direct-expressiveness capability of the DAT-hosting language, this host needs to satisfy the technical criterion of “expressive completeness” [4].

The distinctive technical features of an expressively-complete version of a typical modern programming language is

1. programmer-definable higher-order functions

2. non-strict evaluation
3. symmetric (or “parallel”) implementations of basic logical connectives.

The above requirements 1. and 2. are satisfied by functional languages such as Haskell [5]. Requirement 3. is supportable through an additional parallel existential quantification operator E, such that

$P \perp$	$\exists X_i \in [X_1, \dots, X_n] \cdot P$ $X_i$	$E[X_1, \dots, X_n]$ $P$
$\perp$	$\perp$	$\perp$
$\perp$	<b>False</b>	<b>False</b>
$\perp$	<b>True</b>	<b>True</b>
<b>False</b>	<b>False</b>	<b>False</b>
<b>True</b>	<b>True</b>	<b>True</b>

That is, “E [X1, ..., Xn] P” determines as far as possible whether or not one of the Xi satisfies P. Note especially that when “P  $\perp$ ” (alternatively “**not** (P  $\perp$ )”), necessarily all “P Xi” (alternatively “**not** (P Xi)”) because of the monotonicity of P. The cases in which quantification over an infinite list is in fact computable are thus handled.

The symmetric versions of conjunction and disjunction as used throughout below, are specified:

x	$\perp$	$\perp$	$\perp$	F	F	F	T	T	T
y	$\perp$	F	T	$\perp$	F	T	$\perp$	F	T
$x \wedge y$	$\perp$	F	$\perp$	F	F	F	$\perp$	F	T
$x \vee y$	$\perp$	$\perp$	T	$\perp$	F	T	T	T	T

and are definable from E above:

```

let  $x \vee y = E [x, y] (\lambda x . x)$ 
let  $x \wedge y = \mathbf{not} (E [x, y] (\lambda x . \mathbf{not} x))$ 

```

Another useful symmetric operator that will be used extensively below is M (for match), definable from E:

```

let  $M x_0 [x_1, \dots, x_n] = E [x_1, \dots, x_n] (\lambda x . x = x_0)$ 

```

which matches x0 against xi and returns x0 iff a matching xi = x0 is found. (See APPENDIX A for a

summary of notation used throughout this paper.)

### 3.2. Specifications for DATs and DAT-checking

The semantics of DATs may be specified in terms of how conformance to them is checked. DATs originate with the affinity between assertions and the sets of which they are the characteristic predicates. Set-theoretic combinations of DATs are a natural extension in which the set-membership view of checking conformance to DATs is easily preserved. At the extremity of the development - closure under type-theoretic combinations – the specification can be maintained in set-membership terms.

**3.2.1. DATs as predicates i.e. as sets:** we substantiate our contention that assertions and types are ultimately both predicates and which are objectively indistinguishable from sets. An assertion is a condition upon which the definedness of a computation depends. In the functional context, we say that condition *G* *guards* some expression *X*: “**if G then X else abort**” or in shorthand “*G* ? *X*”. It’s useful however to separate *G* into a predicate *P* and a state to which the predicate applies. When the state is identical to the guarded expression *X*, we have a situation that is analogous to (dynamic) type-checking, which is exposed if we define a further shorthand

$$X :: P = P X ? X$$

The notation is deliberately-suggestive of the popular notation for explicit type-checking in modern functional languages e.g. in Haskell [5] where “*v* :: Nat” requires that ‘*v*’ be of static type “Nat”. The suggestion is that predicates that are too complex to be handled by whatever static typing regime applies can be applied dynamically as DATs. Thus, guarding *X* with the assertion “*P X*” equates to dynamic type-checking of *X* according to *P*. For example, to require that expression *X* satisfies some “*is\_prime*” predicate, it suffices to apply the check “*X* :: *is\_prime*”.

Because of the duality between predicates (and their applications) and sets (and their membership tests), we can re-express the above in terms of set membership as our basic principle of DAT-checking:

**SDP** (Set DAT-checking Principle)

$$X :: P = X \in P ? X$$

For DATs that are represented as sets/predicates, SDP is an implementation as well as a specification. Recall however that the key technical challenge this paper addresses is the treatment of incomputable composite DATs (4.4. and 4.5 below).

**3.2.2. Set-theoretic combinations of DATs:** since predicates and sets are conceptually indistinguishable, then DATs should be composable as sets. At the very least, instead of constructing predicates

$$\text{primes\_or\_evens} = \{n \mid \text{is\_prime } n \vee \text{is\_even } n\}$$

it's more transparent to write

$$\text{let primes\_or\_evens} = \text{is\_prime} \cup \text{is\_even}$$

Defining set-theoretic operators on characteristic predicate representations of sets is a straightforward exercise in higher-order functional programming, viz.

$$(s1 \cup s2) x = s1 x \vee s2 x$$

According variants of SDP above, which transform set combinations into predicates applicable to SDP are:

**UDP** (Union DAT-checking Principle)

$$T1 \cup T2 = (\lambda x. x \in T1 \vee x \in T2)$$

and

**CDP** (Complement DAT-checking Principle)

$$\neg T = (\lambda x. \text{not } (x \in T))$$

Because unions of type-theoretic compositions of DATs are sufficient to illustrate the advantages of the formal approach, we defer treatment of complements (which with unions form a universal set-theoretic basis) to further work.

**3.2.3. Type-theoretic combinations of DATs:** Just as set-theoretic combinations of basic assertions-as-sets makes sense, so do type-theoretic combinations, thus reflecting equally the dual nature of DATs. We follow the typical type-compositions of modern functional languages i.e. structures such as tuples, lists and mappings (functions). As with set-theoretic compositions, type-structured DATs are also specified by sets, that is predicates relevant to the respective structures. Note however that these predicates are in general incomputable mathematical specifications requiring refinement into effective implementations.

**MDP** (Mapping DAT-checking Principle)

$$T1 \rightarrow T2$$

$$= (\lambda f. \forall (x, y) \in f \cdot (f \text{ needs } x \Rightarrow x \in T1) \wedge y \in T2)$$

MDP reflects a common view of mapping types, with the precaution that the domain DAT check is qualified by the neededness of the argument; it is the lazy/non-strict version of the more abstract (power) set membership test: “ $F \in \wp (T1 \times T2)$ ”. Note that MDP is incomputable because it implies that every pair  $(x, y) \in f$  be checked for conformance to respective  $(T1, T2)$ .

**LDP** (List DAT-checking Principle)

$$[T] = (\lambda l . \forall x \in l \cdot x \in T)$$

all elements of a list “ $l$ ” must conform to the given DAT. By requiring that each  $x \in l$  be checked for conformance to  $T$ , LDP compromises lazy evaluation, and is accordingly incomputable in the case of infinite lists.

**TDP** (Tuple DAT-checking Principle)

$$(T1, \dots, Tn) = (\lambda t . \forall xi \in t \cdot xi \in Ti)$$

respective elements of a tuple “ $t$ ” must conform to the corresponding tuple DAT component. Because tuples are finite, TDP happens to be computable, but as with lists compromises lazy evaluation.

Finally, note that it’s reasonable that just as DATs should be composable as if types, the results of these type-theoretic compositions are still conceptually composable as sets. While this requirement is accommodated in the specification (all the above Principles manipulate and generate mutually-compatible sets/predicates), implementation of these compositions emerges as a challenge below.

#### 4. FORMAL DERIVATION OF DAT-CHECKING IMPLEMENTATION

Goals of the formal derivation of DAT-checking are

- *simplicity*: rather than numerous rules (e.g. one for each DAT constructor and combination thereof), expose their common conceptual foundation
- *soundness*: ensure their correctness against specifications
- *extensive coverage*: an intuitive derivation risks leaving some combinations unaddressed.

For a summary of the complete set of DAT checking rules, see APPENDIX B.

We now proceed to transform the above specifications into implementations, using a combination of mathematical identities augmented by a few basic identifiable compromises required in order to render

the specifications computable. The key difference between our implementation and the specification is as follows. In the latter all DATs are predicates so DAT-checking simply applies the predicate. In the implementation DATs will be represented by a variety of structures (essentially a discriminated union or “variant record”) so DAT-checking will involve case analysis on this structure. Thus, our derivation essentially refines the specification of the DAT-checking operator ‘::’ on DATs as (sometimes incomputable) predicates into an implementation of ‘::’ on DATs as data structures. It’s tempting to hope that only the type-theoretic compositions of DATs are “contaminated” thus by incomputability in the specifications, but the mutual closure of DATs under both set- and type-theoretic operations means that straightforward set-theoretic DAT combinations may include incomputable type-theoretic combinations in their subterms.

#### 4.1. Computable Approximation to Infinite Predicates

We propose that the best that can be done with inherently incomputable predicates is to transform them into computable approximations. In addition to a repertoire of mathematical identities elaborated below, the key component of the derivation is a non-identity (“approximation”) that is necessary in order to transform inherently non-executable specifications (i.e. it’s not just the form of the specification, but what is being specified that’s incomputable) into executable implementations. The advantages of this approach to handling incomputability are that (a) it permits isolation of the approximations in the derivation, and (b) it manifests the approximation that is required. In order to emphasise the distinction, we write  $A \approx B$  (rather than  $A = B$ ) when the transformation from A to B is an approximation rather than an identity.

The approximation concerns the transformation of predicates quantified over entire infinite structures (and so incomputable) into predicates applicable to individual elements (and so computable):

**USD** (Universal structure-guard distribution).

$$\forall x \in S \cdot P x \ ? S \approx \{ P x \ ? x \mid x \in S \}$$

Guarding a structure by an assertion that must hold for all elements of the structure, is approximated by guarding each element separately.

The import of USD is that instead of requiring that all elements of a structure conform to a predicate, only those that are actually evaluated need conform, compatibly with the requirement that guides this specification, namely the operational interpretation of assertions. USD is compatible with lazy evalua-

tion of structures, and the more elements are evaluated the closer the approximation matches mathematical identity. Unevaluated elements that may violate the predicate are the potential source of departure from the logical interpretation, and thus any failure of DAT-checking with respect to this interpretation is a “live failure”.

So, USD will be used judiciously in the derivation of DAT-checking below. Note also that the approximation should not be mistaken as an axiom in an extended logic available for free use. Rather, it serves to identify and label steps in a derivation where mathematical identity is necessarily violated in order to restore computability. Finally, specializations of the generic USD for specific structures (functions, lists, tuples) could be given, but in this presentation we prefer to explain each case as needed in terms of the general rule.

#### 4.2. Identities

The actual derivations will be structured around a number of frequently-used identities, each of which should generally be proveable by case analysis.

**FRF** (Function range focus).

$$\{P(x, y) ? (x, y) \mid (x, y) \in F\} = \{(x, P(x, y) ? y) \mid (x, y) \in F\}$$

The dependency of the definedness of an argument-result pair in a function on some guard can be transferred to guarding just the result, because undefining the result for that argument is equivalent to removing argument-result pair from the function.

**GEI** (Guarded equality introduction).

$$P1 \vee P2 ? X = ((P1 ? X) = X) \vee ((P2 ? X) = X) ? X$$

the disjuncts in a guard on an expression can be rewritten as individual tests that each guard succeeds.

**CGS** (Conjunctive guard sequentialisation).

$$P1 \wedge P2 ? X = P1 ? (P2 ? X)$$

A conjunctive guard on an expression can be rewritten as a sequence of applications guards.

**GFE** (Guarded function extension):

$$(P ? F) X = P ? (F X)$$

Application of a guarded function is equivalent to guarding the function application.

**PRC** (Pair-rule correspondence):

$$\{ (x, y) \mid (x, y) \in F \} = (\lambda x. F x)$$

A definition of a function as a set of ordered pairs has an equivalent definition by a computation rule.

**RIE** (Rename image element).

$$\{ (x, F1 y) \mid (x, y) \in F2 \} = \{ (x, y') \mid (x, y') \in F1 \circ F2 \}$$

Applying a function to the result of a function is the same as function composition.

### 4.3. Deriving Simple Predicate DAT Checks

Predicates remain trivially-checkable. In the case that the DAT is simply a predicate applicable to the object of the check, SDP equally simply applies directly:

$$X :: T = X \in T ? X$$

as per both the informal presentation and the formal specification above (SDP).

### 4.4. Deriving Type-composition DAT Checks

Type structures' derivations use functions as the model. For the different kinds of type-composed DATs, variants of '::' are derived using an approximation as follows.

**4.4.1. Functions:** because the predicate for a mapping DAT is not computable (MDP), a computable approximation (USD) must be applied.

$$F :: T1 \rightarrow T2$$

= (SDP, applicable in general)

$$F \in T1 \rightarrow T2 ? F$$

= (MDP, characteristic of this case)

$$(\forall (x, y) \in F \cdot (F \text{ needs } x \Rightarrow x \in T1) \wedge y \in T2) ? F$$

≈ (USD approximation, because the guard on F is inherently incomputable)

$$\{ (F \text{ needs } x \Rightarrow x \in T1) \wedge y \in T2 ? (x, y) \mid (x, y) \in F \}$$

= (FRF, concentrate guard on function result)

$$\{ (x, (F \text{ needs } x \Rightarrow x \in T1) \wedge y \in T2 ? y) \mid (x, y) \in F \}$$

= (RIE, combine operation of function and result guard)

$$\{ (x, y') \mid (x, y') \in (\lambda y. (F \text{ needs } x \Rightarrow x \in T1) \wedge y \in T2 ? y) \circ F \}$$

= (PRC, re-express combination as computation rule)  
 $(\lambda x. ((\lambda y. (F \text{ needs } x \Rightarrow x \in T1) \wedge y \in T2 ? y) \circ F) x)$   
 = ( $\circ$ , simplify function compositions)  
 $(\lambda x. (F \text{ needs } x \Rightarrow x \in T1) \wedge F x \in T2 ? F x)$   
 = (CGS, permits separate operationalisation of conjuncts in guard)  
 $(\lambda x. (F \text{ needs } x \Rightarrow x \in T1) ? (F x \in T2 ? F x))$   
 = (SDP, gives single occurrence of “F x”)  
 $(\lambda x. (F \text{ needs } x \Rightarrow x \in T1) ? F x :: T2)$   
 = ...

At this point, we observe that the requirement “F needs  $x \Rightarrow x \in T1$ ” can be implemented simply by applying the guard “ $x \in T1$ ” to  $x$ , which will inherently only be evaluated if  $x$  is needed by  $F$

= (resuming derivation)  
 $(\lambda x. F (x \in T1 ? x) :: T2)$   
 = (SDP, to the form of the earlier informally-derived result)  
 $(\lambda x. F (x :: T1) :: T2)$

This result implements the operational interpretation of assertions, that is to apply  $T1$  only to the arguments to which  $F$  is applied, and  $T2$  only to the corresponding results. Importantly, it also respects any laziness in  $F$  (the checked argument “ $x :: T1$ ” is only evaluated and checked if  $F$  needs it).

**4.4.2. Other structures:** lists and tuples imitate functions. Rather than derive independently DAT checking for these other structures, it’s conceptually efficient instead to view them as functions from some Index domain to their elements.

For lists, specification LDP obviously compromises laziness applying checking to each element (a reflection of the incomputability in DAT checking for mappings as specified by MDP. Instead, it is preferable to apply the DAT check equally lazily, to each element. The derivation to achieve this proceeds:

$[X1, \dots, Xn] :: [T]$   
 = (lists as functions from above)  
 $[X1, \dots, Xn] :: \text{Index} \rightarrow T$   
 $\approx$  (checking function DATs, as above)  
 $(\lambda i. [X1, \dots, Xn] (i :: \text{Index}) :: T)$   
 = (since in concept  $[X1, \dots, Xn] i :: T = Xi :: T$ )

$$\begin{aligned}
& (\lambda i . [X1 :: T, \dots, Xn :: T] (i :: \text{Index}) ) \\
& = \text{(checking } i :: \text{Index is superfluous, as would be handled by subscripting mechanism)} \\
& (\lambda i . [X1 :: T, \dots, Xn :: T] i) \\
& = \text{(eta-conversion)} \\
& [X1 :: T, \dots Xn :: T]
\end{aligned}$$

The formal derivation of the corresponding rule for tuples

$$(X1, \dots, Xn) :: (T1, \dots, Tn) \approx (X1 :: T1, \dots, Xn :: Tn)$$

would seem to follow accordingly in principle.

#### 4.5. Deriving Set-composition DAT Checks

Likewise different variants of ‘::’ exist for set-theoretic compositions of DATs. Recall that for the purposes of this introduction of our approximations method, we derive union only and leave complement for later treatment.

**4.5.1. Basic unions:** for  $T_i$  in which “ $X \in T_i$ ” is computable, we could render “ $X :: T1 \cup T2$ ” through union of  $T_i$  as per UDP. However, it transpires that an equivalent formulation in terms of the above M operator is more useful (specifically in the derivation of unions of mappings) below.

Thus, checking a union of DATs entails parallel checking of each of the united DATs. The simplest/default case is when the DATs  $T1, T2$  cover simple data values which can be compared for equality using M (and for which we expect  $T_i$  to be computable predicates):

$$\begin{aligned}
& X :: T1 \cup T2 \\
& = \text{(SDP, applicable in general)} \\
& X \in T1 \cup T2 ? X \\
& = \text{(UDP, characteristic of this case)} \\
& X \in T1 \vee X \in T2 ? X \\
& = \text{(GEI, allows ultimate independent operationalisation of guard disjuncts)} \\
& ((X \in T1 ? X) = X) \vee ((X \in T2 ? X) = X) ? X \\
& = \text{(SDP, simplifying)} \\
& (X :: T1 = X) \vee (X :: T2 = X) ? X \\
& = \text{(M, parallel/symmetric match operator)}
\end{aligned}$$

$$M X [X :: T1, X :: T2]$$

as per the informal presentation above. Evaluation of the “ $X :: Ti$ ” would now proceed and against the result of which  $X$  could be matched by  $M$ .

This modified specification is a computable implementation provided that the equality test implicit in  $M$  is defined on  $X$ . It would seem that definedness of equality is effectively synonymous with computability of predicates, at least as in our system (i.e. characteristic predicates augmented by set-and type-theoretic compositions). We now proceed to handle cases where this equality is not defined.

**4.5.2. Unions of mappings:** checking a union of function/mapping DATs likewise entails parallel checking of each of the united DATs, but in the context of actual function application.

In the treatment of simple unions above, for  $X$  over which ‘=’ is defined (and for which we expect the corresponding  $T$  to be computable predicates), the result would be complete. However, for other  $X$ ,  $T$  (archetypally functions  $F$  and DATs  $Ti = Ti.1 \rightarrow Ti.2$ , but also for structures in general), an extended derivation is required. The essence of the problem is: when we have DATs, the implementations for which are computation rules that only approximate incomputable specifications, how do we faithfully apply the semantics of some combination (specifically union) to these implementations?

$$F :: T1.1 \rightarrow T1.2 \cup T2.1 \rightarrow T2.2$$

= (SDP, applicable in general)

$$F \in T1.1 \rightarrow T1.2 \cup T2.1 \rightarrow T2.2 ? F$$

= (eta-conversion, makes guarded function argument explicit)

$$(\lambda x . (F \in T1.1 \rightarrow T1.2 \cup T2.1 \rightarrow T2.2 ? F) x)$$

= (GFE, directly associates argument with guarded function)

$$(\lambda x . F \in T1.1 \rightarrow T1.2 \cup T2.1 \rightarrow T2.2 ? F x)$$

= (UDP, characteristic of this case)

$$(\lambda x . F \in T1.1 \rightarrow T1.2 \vee F \in T2.1 \rightarrow T2.2 ? F x)$$

= (GEI, allows ultimate independent operationalisation of guard disjuncts)

$$(\lambda x . ((F \in T1.1 \rightarrow T1.2 ? F x) = F x) \vee ((F \in T2.1 \rightarrow T2.2 ? F x) = F x) ? F x)$$

= (GFE, conveniently disassociates argument from guarded function)

$$(\lambda x . ((F \in T1.1 \rightarrow T1.2 ? F) x = F x) \vee ((F \in T2.1 \rightarrow T2.2 ? F) x = F x) ? F x)$$

= (SDP, twice)

$(\lambda x . ((F :: T1.1 \rightarrow T1.2) x = F x) \vee ((F :: T2.1 \rightarrow T2.2) x = F x) ? F x)$

At this point, we see that because DAT-checking behaves as a retraction, that is it either leaves a value unchanged (check succeeds) or forces abortion (check fails), then we can use the “parallel match” operator M as highlighted above as the essence of a solution.

= (M, parallel/symmetric match operator)

$(\lambda x . M (F x) [(F :: T1.1 \rightarrow T1.2) x, (F :: T2.1 \rightarrow T2.2) x])$

That is, to check that a function F is of one or another mapping DATs “ $(T1.1 \rightarrow T1.2) \cup (T2.1 \rightarrow T2.2)$ ”, separately apply each mapping DAT to F, and compare the result of applying the checked versions of F to the original unchecked versions. If one of the checked versions gives the same result as the unchecked F, then the union of mapping DATs is satisfied.

**4.5.3. Unions of higher-order mappings:** unions of higher-order, specifically function-valued function DATs accordingly entail checking of the united DATs in the context of the application of the function resulting from the application of the original function etc. First, note that in the immediately-previous derivation for unions of mappings, we could have proceeded alternatively:

$F :: T1.1 \rightarrow T1.2 \cup T2.1 \rightarrow T2.2$

= (basic unions)

$M F [F :: T1.1 \rightarrow T1.2, F :: T2.1 \rightarrow T2.2]$

Comparison with the actual result for unions of mappings allows us to enunciate a new identity:

**EMR** (eta-rule for M operator in context of retracts):

$M F [F :: T1.1 \rightarrow T1.2, F :: T2.1 \rightarrow T2.2]$

=  $(\lambda x . M (F x) [(F :: T1.1 \rightarrow T1.2) x, (F :: T2.1 \rightarrow T2.2) x])$

when “ $F :: Ti.1 \rightarrow Ti.2$ ” typify retracts of F since for any X, either “ $(F :: Ti.1 \rightarrow Ti.2) X = X$ ” or “ $(F :: Ti.1 \rightarrow Ti.2) X = \perp$ ”. EMR allows that in the case that when “ $F :: T1.1 \rightarrow T1.2 \cup T2.1 \rightarrow T2.2$ ” is implemented by “ $(\lambda x . M (F x) [(F :: T1.1 \rightarrow T1.2) x, (F :: T2.1 \rightarrow T2.2) x])$ ”, but “F x” results in a function for which equality is not defined (typifying a structure containing a structure and thus M not directly applicable), then further application of “F x” towards a ground result is permissible:

$$\begin{aligned}
& (\lambda x . M (F x) [(F :: T1.1 \rightarrow T1.2) x, (F :: T2.1 \rightarrow T2.2) x] ) \\
& = \\
& (\lambda x, x' . M ((F x) x') [((F :: T1.1 \rightarrow T1.2) x) x', ((F :: T2.1 \rightarrow T2.2) x) x'])
\end{aligned}$$

That is, we can cover the cases when F is higher-order:

- when application of F *results* in another function (i.e.  $Ti.2 = Ti.2.1 \rightarrow Ti.2.2$ ), or equivalently when a structure contains another structure, proceed as above;
- when F *applies* to another function (i.e.  $Ti.1 = Ti.1.1 \rightarrow Ti.1.2$ ) however, no special treatment is necessary since application of this check to the operand of F is already provided for in existing rules.

**4.5.4. Unions of other structures:** lists and tuples continue to follow the pattern set by functions. Because lists and tuples are viewable as functions from some index domain to their elements, unions of these structures follow the above solutions for function unions.

Thereby, first we establish the rules for unions of lists:

$$Xs :: [T1] \cup [T2] \approx Xs :: [T1 \cup T2]$$

and unions of tuples:

$$\begin{aligned}
& (X1, \dots, Xn) :: (T1.1, \dots, T1.n) \cup (T2.1, \dots, T2.n) \\
& \approx (X1, \dots, Xn) :: (T1.1 \cup T2.1, \dots, T1.n \cup T2.n)
\end{aligned}$$

Unions of list and tuple DATs follow the precedent set individually. Just as the treatments of list and tuple DATs alone involve distributions of DAT checks lazily over individual elements, so accordingly do unions of list and tuple DATs ultimately apply to individual elements.

Second, these rules should respectively generalise to higher-order structures (structures of structures e.g. lists of lists, lists of tuples, lists of functions, lists of lists of etc., tuples of etc.) by way of treating the various  $Ti$  above as mappings  $Ti.1 \rightarrow Ti.2$ . For example:

$$Xss :: [[T1]] \cup [[T2]] \approx Xss :: [[T1 \cup T2]]$$

For illustration, we show this in detail:

$$\begin{aligned}
& Xss :: [[T1]] \cup [[T2]] \\
& = \text{(treating a list as a function over an index)} \\
& Xss :: \text{Index} \rightarrow [T1] \cup \text{Index} \rightarrow [T2]
\end{aligned}$$

= *(function unions)*

$(\lambda i . M (Xss\ i) [(Xss :: Index \rightarrow [T1])\ i, (Xss :: Index \rightarrow [T2])\ i])$

$\approx$  *(functions)*

$(\lambda i . M (Xss\ i) [Xss\ (i :: Index) :: [T1], Xss\ (i :: Index) :: [T2]])$

= *(effecting subscripting, Xs is i-th element of Xss, in the course of which i :: Index is inherently effected)*

$(\lambda i . M\ Xs\ [Xs :: [T1], Xs :: [T2]])$

= *(again, treating a list as a function over an index)*

$(\lambda i . M\ Xs\ [Xs :: Index' \rightarrow T1, Xs :: Index' \rightarrow T2])$

= *(EMR)*

$(\lambda i, i' . M\ Xs\ [(Xs :: Index' \rightarrow T1)\ i', (Xs :: Index' \rightarrow T2)\ i'])$

$\approx$  *(functions)*

$(\lambda i, i' . M\ Xs\ [Xs\ (i' :: Index') :: T1, Xs\ (i' :: Index') :: T2])$

= *(effecting subscripting, X is i'-th element of Xs, in the course of which i' :: Index' is inherently effected)*

$(\lambda i, i' . M\ X\ [X :: T1, X :: T2])$

= ...

Thus, the element selected from Xss must conform to either T1 or T2; in other words:

= *(resuming)*

$Xss :: [[T1 \cup T2]]$

## 5. CONCLUSIONS

Both DATs and our approach to their derivation seem beneficial in themselves and with potential for further development.

### 5.1. Benefits of Approach to Formal Derivation

The primary contribution of this paper is that it illustrates a method by which a problem in the integration of types and assertions can be managed, that is handling the inherent incomputability of the predicates formed by closure under both set- and type-theoretic operations. The key to the solution is to isolate a single “approximation” rule that can be used in deriving implementations. As well as flagging clearly the use of the approximation, it indicates where further attention may profitably be directed in improving the result.

In this specific context (DATs), the effect of the approximations on the derived implementation can be understood in terms of the difference between the logical and operational interpretations of assertions: the former requires an assertion to apply for every possible execution, whereas the latter implements the former by testing only for every actual execution. Our “approximate” implementation is thus entirely consistent with the operational interpretation, i.e. weakening the specification so as to test only every actual execution (i.e. member of the structure that is actually used). Obviously, the lazy-evaluation properties of functional languages prove crucial in this regard. Further, it’s significant that the formal derivation identified and validated the implementation of an aspect of DATs (unions of structures of structures etc.) that an earlier informally-derived version ignored [6].

## 5.2. Benefits of DATs

While the focus is on the means by which DATs have been achieved, what they offer is a useful facility in summary as follows.

- DATs are first-class objects: they can be passed as parameters to any component, and can be arbitrarily parameterized themselves (i.e. full generic capability).
- DATs can be specified to the precision of arbitrary computable sets, compared to static types. For example, we can characterize such as the set of primes as a dynamic type.
- DATs provide familiar and useful means (set- and type-theoretic) of composing assertions.
- DAT-checking can be applied to arbitrary expressions, not just declared variables/parameters/functions.
- Most of the DAT system as informally derived [6] has been implemented [7] as an extension of the “Refine” prototyping language [8]. The extensions discovered by the formal derivation should be equally implementable in such a weakly-typed host.
- The correctness of DAT-checking is exposed by the formal derivation, and approximations isolated.

## 5.3. Further Directions?

As well as the possibility of applying our technique of isolating inequalities in other derivations besides DATs, several DAT-specifics suggest themselves.

**5.3.1. Completion and extension of closure:** it’s tempting to consider heterogeneous combinations (e.g. lists united with atomic data) and adaptation of the above for more flexible type systems, or even an untyped environment, does seem possible given the fact that our unions are based on generally applicable computational rules. However, our first priority is to complete the treatment within a statically-typed host environment. Just as the formal derivation aided the discovery of additional DAT checking

rules for closure of unions of higher-order structures – functions, lists and tuples whose results/contents are structures – we hope that continued work will lead to more complete coverage i.e. to handle complements of unions of type-structured (functions, lists, tuples) DATs. as well as more accurate approximations.

**5.3.2. Direction in semantic modeling:** as we have several times acknowledged, from a conceptual point of view DATs are retraction mappings. [9]. However, in the above presentation, DATs are represented as data structures that are in effect interpreted by the ‘::’ operator to give the behaviour of retracts. An overall simpler presentation should result from direct actual representation of DATs as retraction mappings, which the expressively-complete, higher-order functional host should permit.

**5.3.3. Parallel implementations of functional languages:** we acknowledge that the general potential inefficiency of dynamic type-checking is if anything exacerbated by the necessity for parallelism in the implementation. However, it’s our conclusion that this application demonstrates that parallel operations are not just an option by a necessity for the realization of the full potential of functional programming in terms of expressive completeness, and that the quest for efficient parallel implementations of functional languages needs to continue.

## 6. ACKNOWLEDGEMENTS

This paper is an extensive development of [6], and we are grateful to Bailes’ earlier co-authors Ming Gong and Andrew Moran for their contributions to this effort, which has been supported by the Australian Research Council.

## REFERENCES

- [1] Tennent, R.D., “Principles of Programming Languages”, Prentice-Hall (1981).
- [2] Meyer, B., “Object-oriented Software Construction”, Prentice-Hall (1988).
- [3] IEEE, “Special Issue on Formal Methods”, IEEE Software, vol. 7, no. 5 (1990).
- [4] Plotkin, G.D., “LCF Considered as a Programming Language”, Theor. Comp.Sci., vol. 5, pp. 223-255 (1977).
- [5] Thompson, S.J., “Haskell: the Craft of Functional Programming”, Addison-Wesley (1999).
- [6] Bailes, P.A., Gong, M. and Moran, A., “Why Functional Languages Really Need Parallelism”, Proc. ICCI 1993, Sudbury, pp.423-427, IEEE (1993).

[7] Bailes, P.A., Chapman, M., Gong, M. and Peake, I., “GRIT: an Extended Refine for More Executable Specifications”, Proceedings 8th KBSE Conference, Chicago, pp. 123-132, IEEE (1993).

[8] Reasoning Systems, “Refine User's Guide”, Palo Alto (1992).

[9] Scott, D., “Data Types as Lattices”, SIAM J. Comput. vol 5, no. 3, pp. 522-587 (1976).

## APPENDIX A – Notation and Vocabulary

Mathematical and functional language notation used in this paper follows standard conventions, with concrete syntax in particular following the lambda-calculus/Miranda/Haskell tradition. The following syntactic metavariables appear in the above (ambiguities are resolvable by context):

- E - existential quantification operator
- F - functions
- G - guards (boolean expressions)
- M - match operator
- P - predicates (boolean-ranged functions)
- S - structures (generally)
- T - DATs
- X - expressions/values
- Xs, Xss - list, list of lists

## APPENDIX B – Summary of DATs and Checking

Basic principles that are directly applicable when DATs T are sets:

$$X :: T = X \in T ? X = \text{if } X \in T \text{ then } X \text{ else abort}$$
$$X :: T1 \cup T2 = M X [X :: T1, X :: T2]$$

Approximations for functions and other structures are possible because of **USD** approximation (section 4.1):

$$F :: T1 \rightarrow T2 \approx (\lambda x . F (x :: T1) :: T2)$$
$$[X1, \dots, Xn] :: [T] \approx [X1 :: T, \dots, Xn :: T]$$
$$(X1, \dots, Xn) :: (T1, \dots, Tn) \approx (X1 :: T1, \dots, Xn :: Tn)$$

Unions of mappings are a special case of **EMR** (section 4.5.3):

$$F :: (T1.1 \rightarrow T1.2) \cup (T2.1 \rightarrow T2.2) = (\lambda x. M (F x) [(F :: T1.1 \rightarrow T1.2) x, (F :: T2.1 \rightarrow T2.2) x])$$

Unions of other structures are special cases of unions of mappings:

$$Xs :: [T1] \cup [T2] \approx Xs :: [T1 \cup T2]$$

$$(X1, \dots, Xn) :: (T1.1, \dots, T1.n) \cup (T2.1, \dots, T2.n) \approx (X1, \dots, Xn) :: (T1.1 \cup T2.1, \dots, T1.n \cup T2.n)$$

Because **EMR** is further applicable to unions of mappings when  $T_{i.2}$  are themselves functions (i.e. “ $F x$ ” yields a function), then in particular, structures of structures are handled, e.g.

$$Xss :: [[T1]] \cup [[T2]] \approx Xss :: [[T1 \cup T2]]$$

etc.