Automatically Solving Simultaneous Type Equations for Type Difference Transformations that Redesign Code*

Ted J. Biggerstaff

Software Generators, LLC
dslgen at softwaregenerators dot com

Abstract. This paper introduces a generalization of programming data types called *Context Qualified Types* (or *CQ Type*s for short). CQ Types are a superset of programming language data types. They incorporate design features or contexts that fall outside of the programming data type domain (e.g., a planned program scope). CQ Types are functionally related to other CQ Types (and eventually to conventional data types) such that a differencing operation defined on two related types will produce a program transformation that will convert a computational instance (i.e., code) of the first type into a computational instance of the second type. Large grain abstract relationships between design contexts may be expressed as simultaneous type equations. Solving these equations, given some starting code instances, produces transformations that redesign the code from one design context (e.g., a payload context) to a different design context (e.g., a "hole" within a design framework from a reusable library).

Keywords: Type differencing, context qualified types, CQ Types, design frameworks, design features, functionally related types, transformations, domain driven instantiation, simultaneous type equations.

1 Overview

1.1 The Problem

This paper addresses the problem of automatically integrating two separately derived pieces of code that are potentially compatible but differ in a few design details that will have to be synchronized to achieve compatibility[1]. These design differences arise from slightly differring requirements for the differing specifications. For example, consider a code/design framework (i.e., a combination of skeletal code plus holes

---

* Patent 8,713,515 [7] and patent pending.

[1] This work is a part of the DSLGen™ program generation system, the details of which are beyond the scope of this paper. A more complete discussion of DSLGen™ can be found in [6], as well as other documentation available at www.softwaregenerators.com.

designed to accept insertion of foreign, target computation code, e.g., Ref.[2] 1-01) as one of those pieces of code. Let's call this framework the "Thread Design Framework" (or TDF for short). The TDF framework example expresses a pattern of parallel computation using threads to implement parallel execution but knows virtually nothing about the code to be executed within those threads (i.e., the code in the "framework holes"). Its design might be constrained by the requirements of the thread package used. For example, a planned user (or generator) written routine for initiating a thread might only allow one user data parameter to be passed to the application programmer's thread routine (along with the thread specific parameters, of course). If that thread routine needs more than a single item of user data to operate (e.g., matrices, the dimensions of the matrices, and perhaps some start and end indexes specific to the algorithm), then the programmer (or generator) may have to formulate some custom "glue" code to connect the target computation data to the holes in TDF. For example, he could setup global variables to communicate that data to the thread routine. Alternatively, the programmer could write some "glue" code to package up the set of needed data into a structure (for example) before calling the thread routine, send a pointer to the structure to the thread routine as the single user argument and then unpack the data items within the application's thread routine. In this latter case, he will also have to adapt (i.e., redesign) elements of his "vanilla" payload computation to fit the details of this glue code.

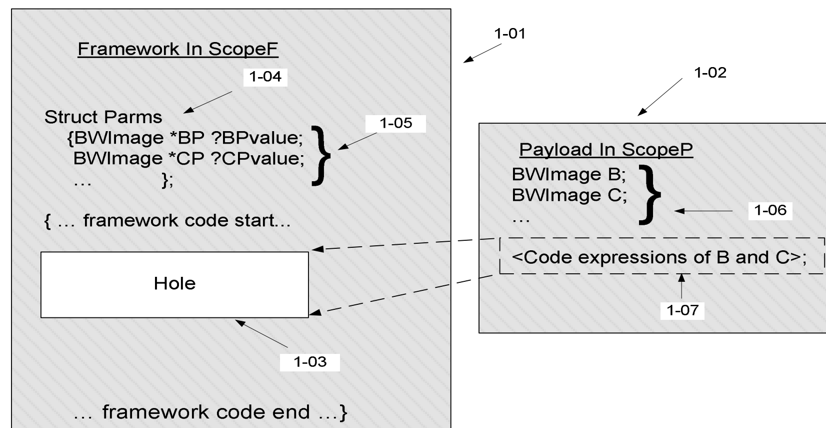

Fig. 1. Skeletal design framework and a code payload

Given those requirements on the thread based code framework (TDF), consider the nature of the target code payload (e.g., Ref. 1-02) that is to be merged into the TDF. That target computation payload might benefit from parallel computation but will require design modifications to execute correctly with the TDF framework (Ref. 1-

[2] Ref. will be use to designate callouts in figures. Ref. *n-mm* will be the *mm* callout in Fig. *n*.

01). For example, among the modifications that will have to be made to synchronize it with the code in the TDF framework is the redesign of the references to its data (e.g., the matrices, etc.) so that they synchronize with the design implicit in TDF's (skeletally defined) thread routine (Ref. 1-01). As mentioned earlier, direct reference via global variables to the matrices and related data (e.g., indexes) is one possible solution but that may require some scope adaptation to both the payload and the framework to allow sharing of the same scope. However, such changes can be daunting for an automatic generation system because a generator has to first model the global scoping structure, which can be challenging. If the framework design cannot be changed (e.g., it is a reusable library component or it calls unchangeable library routines), then the design of the payload will have to be altered by writing some glue code that packages up references to the payload variables and modifies the payload code to connect through the glue code. The glue code option has the benefit of greater localization of the changes thereby making them easier for an automated generation system. This invention provides an automated method for the class of redesign process that allows the payload code to be redesigned so that it can be relocated into the context of a framework (e.g., TDF). For some background on alternative concepts that are similar to but not exactly the same as design frameworks, see [10, 13].

To create the redesign machinery, this work extends the conventional notion of data types by incorporating into the type specification, design features and contexts (e.g., generator-time design features) that fall outside the domain of programming language data types. Thereby the generator is able to define and work directly with features and contexts of an evolving *design abstraction* (i.e., an abstraction *not* based on programming language structures or abstractions thereof). The extended types are called *Context Qualified Types* or *CQ Types.* Additionally, the machinery provides a key mechanism for defining composite *CQ* Types such that there is an *explicit functional relationship* between pairs of CQ Types as well as between programming language data types (e.g., BWImage) and related CQ types (i.e. a pointer to BWImage). This explicit functional relationship determines how to automatically generate *type difference transformations* that can convert an instance (e.g., "B" as in Ref. 1-06) of one of the types (e.g., BWImage) into an instance (e.g., "(& B)") of the other type (e.g., Pointer to BWImage), where "&" is the "address" operator. Also, any two functionally related CQ Types allow the automatic generation of an inverse transformation[3]. The inverse transform will convert an instance of a pointer to a BWImage (e.g., a structure field named "BP") into the BWImage itself (e.g., "(* BP)"). BP might be within a different scope altogether from B (e.g., the "*BP" field in the

---

[3] Many difference transformations are not "pure" functions and may therefore introduce existential variables. For example, an instance of a 2D matrix being transformed into a reference to an item in that matrix may introduce one or more programming language index variables such as ?Idx1 and ?Idx2 whose eventual target program identities (e.g., i and j) may not yet be determined. The mechanisms for dealing with such existential variables and their bindings are beyond the scope of this paper but suffice it to say, DSLGen deals with these problems using domain specific knowledge (see the description of the ARPS protocol below) to stand in for those identities until they can be concretely determined later in the generation process.

framework scope shown as Ref. 1-05). The automatically generated type difference transformations as a class are called *redesign transformations* or *redesigners* for short. This machinery will be used to redesign instances of generated code (e.g., an expression to compute a convolution of an image) so that they can be used to instantiate a partially complete code framework such as TDF (i.e., a pattern of some code and some receptacles for foreign code). Since the framework may need to use data items that are decorated with design features (e.g., a field containing a pointer to the data) that are not a part of the original data item, the original data item will need to be redesigned to synchronize with the requirements of the framework before it can be used in a context of the framework.

A key problem in using code components developed independently in one context (e.g., payload scope) without knowledge of their potential use in and connections to elements of a disparate context (e.g., framework scope) is that there is no feasible way to directly connect the elements of the one context (e.g., a payload context) to the conceptually corresponding elements of the other context (e.g., a framework context) without negating the independence of the two contexts and thereby negating the combinatorial reuse value of combining many independent contexts. Identifying the correspondences between disparate code components by explicit names is not feasible because the two contexts are developed independently and likely, at different times. Parameter connections are not feasible because like explicit naming, this would require some *a priori* coordination of the structure of the two disparate code components, which is not desirable. What the two contexts may know about each other is indirect. It is their domain specific entities, features and topology, which opens the door to a reference protocol that is based on one context searching for known or expected domain specific entities, features and relationships within the other context. This is a key mechanism of this invention. It provides the machinery for expressing "anaphoric" references (i.e., references that are implied) in one code entity (e.g., the framework) to data items (e.g., an image data item that is being used as the output data item for a computation) in a separate, disparate and as yet undefined code entity (e.g., a computational payload to be used within the framework). This mechanism is called the *Anaphoric Reference Protocol for Synchronization* (*ARPS*). (See [7].) The anaphoric reference mechanism expresses references in terms of semantic (and largely domain specific) abstractions rather than in programming language structural forms or patterns (e.g., loops, code blocks, operators, etc.) or abstractions thereof. For example, a structure field within a framework may need a value from some as yet to be defined computational payload. Semantically, the framework knows only that that value will be an image that is being used as the input image of the payload computation. ARPS provides a domain specific notation whereby that relationship can be expressed in the definition of the field within the framework. ARPS provides machinery such that when a candidate computational payload is eventually identified, that ARPS reference will be used like a search query and will automatically resolve to the specific data item needed by the framework.

Once the ARPS expressions in a framework determine the conceptually corresponding items in the payload, the automated *redesigners* are invoked to redesign those payload items so that they are consistent with the framework design. Then the

payload code (suitably redesigned to synchronize with the framework design) can be directly inserted into the holes of the framework.

2 CQ Types, Their Relationships and Type Differencing

2.1 Overview

A CQ type is a programming language type (e.g., a grayscale image matrix type designated as "BWImage") that has additional *Qualifying Properties* specific to another context (e.g., a design feature, such as the payload scope "ScopeP"). CQ Types allow the generator to express concepts that transcend the strictly programming language domain. That is, the CQ type concepts include generation entities, contexts and operations rather than being strictly limited to programming language entities, contexts and operations. Qualifiers can represent design features that do not yet have any program structure manifestation (e.g., an anticipated but not yet created program scope).

A CQ type is designed to have an explicit functional relationship to other CQ types. Fig. 2 shows the conceptual form of relationships between a programming language type (Ref. 2-07) and a number of abstract CQ types (Refs. 2-08a through 2-10), where the CQ property for this example is the name of a payload or framework scope (e.g., ScopeP or ScopeF). In Fig. 2, CQ Types are shown as ovals and instances of those types as boxes. The type to type relationships are implemented via either subtype/supertype relationships (e.g., Refs. 2-08a and 2-09a) or a cross context mapping relationship (Ref. 2-18) that defines some elective transformational mapping between (in the example) an instance of the payload context and a related instance in the framework context. The transformations between instances of CQ subtypes and supertypes (i.e., type differences) are automatically derived from the two CQ types. Cross context mapping transforms (i.e., those between CQ Types that do not have a subtype or supertype interrelationship) are elective and therefore are custom written by a design engineer at the time the related design framework (e.g., TDF) is created. They are designed only once for a specific reusable design framework (e.g., TDF) but will be used/instantiated many times to generate many different concrete, target programs. In the example of this paper, the mapping relationship is computational equivalence between the end points of the type chain (i.e., between instance P2 and instance F2. That is, execution of the initial computational form (P2) in the payload scope will produce the same result as the execution of a different but operationally equivalent computational form (F2) in the framework scope.

Furthermore, each pair of connected CQ Types (i.e., type/supertype or cross connection) implies two, directionally specific Redesign Transformations that will convert an instance of one of the types into an instance of the other that is one step farther along on the path to computational equivalence. The type/subtype transformations are automatically derivable via so called type differencing operations. The form of the type differencing transformations is deterministically implied by the type constructor operators. By contrast, cross connection transforms are custom created by a domain

engineer. Example cross context mappings include, computational equivalence (as in the example presented), data type casts, design translation options, etc. Because all relationships define explicit functional relationships, the generator can use type differencing to harvest a set of Redesign Transformations (i.e., transformations X1 through X5 in Fig. 2) that carry a payload instance P1 of a programming language type used in the payload context into a computationally equivalent instance F1 of a type used in the framework context. X1 and X6 map between the domain of programming language data types and the CQ Types with the design domain.

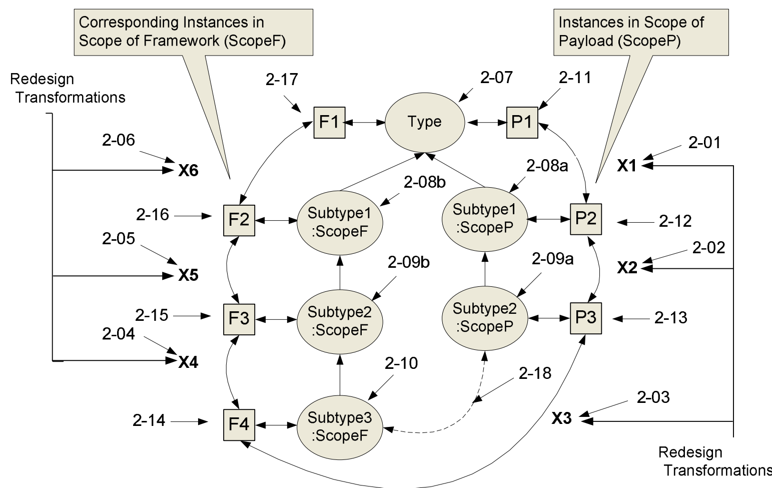

Fig. 2. Chain of CQ Types relating two different design contexts

Fig. 3 provides a concrete example that may be used to solve part of the problem illustrated in Fig. 1. As a debugging aid, CQ Type names are designed to expose both the base programming data type (e.g., "BWImage" or "int") as well as their qualifying property pairs (e.g., ":myscope ScopeF"). The implementation machinery necessitates some special syntax within the type names, specifically, underscores to bind the name parts together and vertical bars to delineate the beginning and ending of the type name. All CQ Types will have "tags" that uniquely identify them (E.g., BWi7, Ptr1 or Field1). These tags may be used in a CQ Type name to reference another CQ Type (e.g., tag "BWi7" in type Ref. 3-02a references its super type Ref. 3-01a).

The instance to instance transformations of Fig. 3 make it clear that subtyping within CQ Types is used to capture subpart/superpart relationships (among others) between design constructs, and thereby it may also imply the construction/deconstruction operations required to achieve transitions between design views and/or design contexts. For example, "B" from the computation specification context (Ref. 3-04) represents the same entity as "B" within the payload scope context (Ref.

3-05) but “(& B)” (Ref. 3-06) is a computational form that represents one step on the pathway to the full computational equivalence finally realized in Ref. 3-09.

Harvesting the transformations implied in Fig. 3 (i.e., X2 through X5) by differencing the CQ Types and applying those transforms to a payload oriented expression like

PartialAns = (B[idx13+(p29 -1)][idx14+(q30 -1)] * w[p29] [q30]) ; (1)

will convert (1) into a design framework context expression like

PartialAns = ((*(rsparms9.BP))[idx13+(p29 -1)][idx14+(q30 -1)]
* w[p29] [q30]); (2)

thereby allowing it to be relocated into a hole in the TDF design framework (e.g., Ref. 1-03). Such relocation assumes, of course, that other payload specific entities (e.g., the start, increment and end values of indexes such as“idx13”, “idx14”, “p29”, etc.) will have to be similarly redesigned.

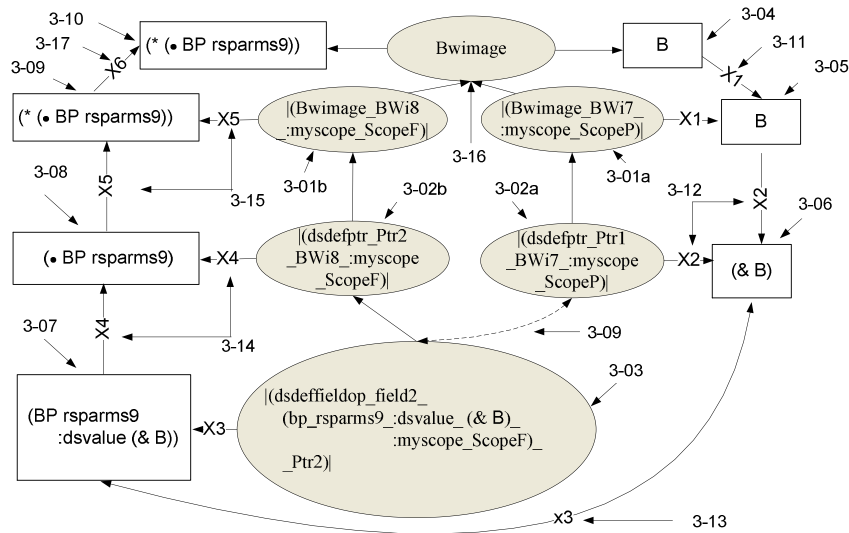

Fig. 3. A concrete example relating framework and payload design contexts

The computational domain of the PartialAns example is the convolution of 2D digital images (where “B” is the image from the example). The deeper programming language design context (i.e., problem specific domain knowledge) for these expressions is as follows: idx13 and idx14 are variables that index over some image matrix B; p29 and p30 are offsets for pixels in a neighborhood around the current pixel

"B[idx13, idx14]"; and w is an array of multiplicative coefficients defining the relative contribution of a neighborhood pixel to the overall magnitude of the PartialAns value. In a complete description of the required redesign operations for this example, analogous CQ Type chains would exist for the start, increment and end values of the indexes idx13, idx14, p29, p30, and possibly other needed data entities. And these would engender analogous conversions for these data entities.

While the example in this paper uses the names B, idx13, idx14, p29 and p30 for concreteness, DSLGen uses an abstract domain specific model for the computation it is operating on. That is to say, it models a convolution in terms of image matrix abstractions, matrix index abstractions, neighborhood abstractions and neighborhood loop index abstractions none of which map into concrete programming language names and entities until very late in the generation process. That is, mapping DSLGen's domain abstractions (e.g., neighborhood loop index abstractions) into concrete names like idx13 and idx14 is deferred until the overall design is complete and the design has stopped evolving, because during that evolution process, the associations between domain abstractions and concrete target names will change to accommodate the evolution of the design. That being said, the author believes that the use of concrete names for this description will aid the reader's understanding of the process and therefore, this paper will continue to use concrete names.

Next, we will define the machinery required to construct Fig. 3 and unlimited numbers of similar constructions from (reusable) parameterized type specifications. Then we will use these constructions to redesign code.

2.2 Solving Simultaneous Parameterized Type Equations

The CQ types of Fig. 3 contain design elements that are custom formulated to produce one set of redesign transformations that are specific to the data entity "B" (because of the concrete field names in type 3-03) within a specific target computation that is to be transformed from a specific payload context to a specific framework context. In order to solve the more general problem that is described in Section 1, other analogous QC Type chains and redesign transformations will have to be developed for other specific data entities in that target computation (e.g., other image variables along with loops' start, end and increment values, e.g., those values for "Idx14"). That is, Fig. 3 must be abstracted and the instantiation of that abstraction automated. It would be a serious impediment to full automation of the generation process to require a human to custom build of each of the individual type structure analogs of Fig. 3 for all of the data entities within a specific computation. Hence, there needs to be a single (reusable) precursor abstraction or abstractions from which all Fig. 3 analogs and the type difference transformation analogs associated with all target computations (including Fig. 3) can be automatically derived. That single precursor abstraction comprises: 1) a set of parameterized type equations (expressions (3)-(7) below) that capture, in a more abstract and reusable sense, the relationships (i.e., type chains) illustrated in Fig. 3 and 2) two special parameterized type difference transformations specific to the TDF framework that express the cross connection mapping between the two type subtrees. One of these TDF framework supplied difference transformations

is shown as expression (9) below. The cross connection transformations are identified in Fig. 3 as Ref. 3-13 (labeled X3). Ref. 3-13 represents both mappings to and from instances of the types 3-02a and 3-03.

The remainder of this section will describe that parameterized precursor specification. It is expressed as a set of simultaneous type equations (expressions (3)-(7) below). This section will further describe the process by which those type equations are incrementally solved to produce concrete CQ Types (Refs. 3-01a & b, 3-02a & b, and 3-03) and simultaneously to produce type difference transformations (Ref. 3-12 through 3-15 in Fig. 3), which are represented more concretely, as expressions (8)-(11) below. In the course of these steps, the type difference transformations are incrementally applied to a starting instance of one of those types (Ref. 3-05) and then to its derivatives (3-06 through 3-08) to derive the correspondence between "B" in scopeP and the equivalent glue code "(* (● BP rsparms9))" in scopeF, which will eventually become the C language code "(*(rsparms9.BP))".

The simultaneous parameterized type equations that will generate Fig. 3 (and all of its analogs for similarly structured problems) are expressions (3)-(7):

```
(?t1  = (?itype :myscope ?Pscope :Instname  '?StartInst)) ;;spec for type 3-01a  (3)
(?t2  = (DSDefPtr  ?t1 :Instname '?PPtrinst))              ;;spec for type 3-02a  (4)
(?t5  = (?itype :myscope ?Fscope))                         ;; spec for type3-01b  (5)
(?t4 =  (DSDefPtr ?t5 :Instname '?FPtrinst))               ;; spec for type 3-02b (6)
(?t3  = (DSDefFieldOp (?Fldname ?Structname :dsvalue ?PPtrinst
                                  :myscope ?Fscope :Instname '?Fldinst)
                                 ?t4 ))                    ;; spec for type 3-03   (7)
```

The *type constructor functions* used in these expressions (i.e., DSDefPtr, DSDefFieldOp as well as the qualified programming language data types of the form "(<*atype*> <*qualifiers*>…)) will be described here informally. This description in combination with the contextual intuition provided by Fig. 3 and the sampling of formal definitional forms of section 3 of this paper should provide sufficient insight to understand the essence of them and the associated process. However, the full set of type constructors available in DSLGen (e.g., DSDef… constructors for Ptr, Field, Struct, Bit, Enum, Union, Function, Array and qualified programming language data types) is described more completely and formally in [7] (Patent No. 8,713,515). Any additional user defined type constructors will follow the same patterns.

Basically, each equation's right hand side is a type constructor expression of the form *(operator …arguments …)*, which specifies a composite CQ Type. Implied functionally related types within these forms are expressed via the variables ?t1, ?t2, etc., and represent recursively related CQ types (often related via type/supertype relationships). For example, the equation (4) of the form "(DSDefPtr ?t1 …)" is a pointer type (which will eventually become type 3-02a). The referenced supertype is represented by the variable ?t1, which is bound to the "(?itype :myscope ?Pscope :Instname '?StartInst)" type specification from equation (3). ?t1 will become the type 3-01a in Fig. 3. The "*:name value*" pairs in these expressions (e.g., ":myscope ?Pscope") are qualifying information used to describe generator design contexts or

features that fall outside of the strict domain of conventional data types. Explanatory comments appear to the right of the double semicolons. The ":Instname '<?vbl>" pairs (omitted in Fig. 3 to save space) provide (*quoted*) simultaneous variable names to the type differencing routines that harvest the *ReDesign* transformations associated with type pairs. These variables provide global relationships among the individual type equations, the related type difference transformations and the instances upon which those difference transformation operate. For example, difference transformation (9) that relates types 3-01a and 3-02a will bind the instance of type 3-02a (e.g., "(& B)") to the transformation variable name "?PPtrinst." That value of ?PPtrinst is later used in the value slot of the field instance (i.e., 3-07) being constructed for the field type 3-03. Expression (9) also contains some embedded Lisp code that will create a human friendly name for the pointer field thereby making the generated code a bit easier to understand. It concatenates ?Pinst's value (e.g., "B") with the letter "P" resulting in the pointer field being named "BP" in the Fscope context (i.e., "BP" in Fig. 3). It then calls "DSRuntimeDeclare" to declare "BP" to be an instance of type ?t1. The C language code generated to reference the BWimage B within the Fscope will be something like "(rsparms9.BP)" (see expression (2)). The results for other needed variables from Pscope (e.g., "Incr-Idx14) will acquire analogous names (e.g., "Incr-Idx14P").

To solve these type equations, we will need to use the type differencing transforms implied by these type relationships since they express the relationships of these CQ Types to legitimate instances of them and thereby, they recursively constrain subsequent CQ Types and instances. For example, consider the type differencing transformation from ?t1 to ?t2, which is symbolically expressed as "(delta ?t1 ?t2)" and which is nominally defined by the transformation expression to the right of the "=" sign in expression (8) below. Within that transformation expression, the sub-expression to the left of the "=>" is the *pattern* of the (delta ?t1 ?t2) transformation. That pattern is an "*AND" pattern* (operationally the Lisp expression "$(PAND ..)"). An AND pattern requires that all of its constituent sub-patterns successfully match some instance of type ?t1. That is, its first pattern element "?inst01" will succeed (because it is initially unbound) by matching an instance of type ?t1 (e.g., "B") and binding that instance to a pattern variable (i.e., "?inst01") where the variable name is uniquely generated by the machinery that generates the type difference transformation. The second pattern element of the AND operator, i.e., "?StartInst", will succeed by binding that same instance to the pattern variable (i.e., "?StartInst"), where "?StartInst" was supplied by the property pair ":Instname '?StartInst" from the specification of type ?t1 shown in expression (3). If the pattern match is successful, then the right hand side of the transformation (i.e., the expression to the right of "=>", which is "(& ?inst01)") will convert ?inst01 to an instance of ?t2 (i.e., "(& B)"). Subsequently, the next type difference (i.e., "(delta ?t2 ?t3)" defined in expression (8)) will bind that newly created instance (i.e., "(& B)") to the variable ?PPtrinst, which will eventually be used in the dsvalue slot of ?t5.

The transformational essences that are the results of differencing the functionally related types in equations (3)-(7) are shown in the expressions (8)-(11). Difference (8) executes the operation that is defined by the DSDefPtr constructor of type equation

(4). Similarly, the differences (10) and (11) just express the *implied inverses* of the type constructors DSDefFieldOP and DSDefPtr of type equations (7) and (6). Difference (9) is the cross context mapping relationship and is supplied by the design framework TDF in the generator's library. The TDF framework is built specifically for generating the adaptive "glue code" that we have been discussing up to this point.

```
((Delta ?t1 ?t2) ~ ($(pand ?inst01 ?StartInst) =>
                        (& ?inst01))) ;; ?inst01 is globally unique name          (8)

((Delta ?t2 ?t3) ~ ;; Cross Connection written by a Design Engineer for TDF
   ($(pand ?PPtrInst $(plisp (MakeBinding '?Fldname
                        (DSRuntimeDeclare (quote ?t1) (symb ?StartInst 'P))))) )
      => (?Fldname ?Structname :dsvalue ?PPtrInst))) ;;Field Def. form          (9)

((Delta ?t3 ?t4) ~  ;; Field def. form to dot operator
   ($(pand ?Fldinst (?fld ?struct :dsvalue ?PPtrInst)) =>  (● ?fld ?struct)))     (10)

((Delta ?t4 ?t5) ~  ;; Dereference pointer result of dot operation
   ($(pand ?inst02 ?FPtrinst) => (* ?inst02)))                                    (11)
```

Operationally, these type Delta transformations are implemented by ReDesign multi-methods[4] that are uniquely determined by the method name ("ReDesign) plus their first two arguments, type1 and type2. ReDesign methods also take additional arguments: 1) an instance of type1, and 2) an initial set of bindings. The initial bindings include bindings determined by the TDF framework context (e.g., the bindings "(?itype bwimage)" and "(?structname rsparms9)") ; bindings that arose during type equation solution process (e.g., "(?t5 |(bwimage_bwi8_:myscope_scopef)|)") ; and bindings created by previously processed ReDesign steps (e.g., "(?startinst B)").

So, what does a ReDesigner look like? Expression (12) is some CommonLisp like pseudo-code that specifies the essence of a ReDesigner's processing.

```
(ReDesign  type1  type2  instance  bindings) ::=
     (let ((newinstance  nil))
          (multiple-value-bind (success  postmatchbindings )
               (match (LHS (Delta type1 type2))  instance  bindings)
               (if success (setf newinstance
                         (applysubstitution (RHS (Delta type1 type2))
                                        postmatchbindings)))
               (values success postmatchbindings newinstance)))                 (12)
```

[4] These multi-methods are expressed in the CLOS (CommonLisp Object System) language embedded in CommonLisp. They are automatically generated during the process that solves the type equations for concrete types. The single exceptions to automatic generation are any cross type differencing methods (e.g., expression (9)), which are written by the Design Engineer at the time the design framework (e.g., TDF) is created and entered into the reusable library of frameworks. They express *elective* design mappings between CQ types.

In (12), the CommonLisp *multiple-value-bind* operator defines a scope with two local Lisp variables (i.e., success and postmatchbindings) to receive the multiple values returned from the match routine, which matches the left hand side (LHS) pattern of the specific type Delta transformation against the instance argument. The match starts using the existing bindings in the variable "bindings". If the match is successful (i.e., success equals t on match's exit), then the new instance will be the right hand side (RHS) of the Delta instantiated with the bindings returned from match, i.e., postmatchbindings, which are the initial bindings extended by any new bindings created by the match routine (e.g., "(?startinst B)"). The ReDesigner returns three values: 1) the success flag, 2) the postmatchbindings and 3) the newinstance, where the latter two variables will have legitimate values only if success equals t.

Before the type equations are solved, a typical set of initial bindings supplied by the TDF setup code might be:

((?itype bwimage) (?pscope scopep) (?fscope scopef) (?structname rsparms9)) (13)

After the type equations are solved creating a set of types like those shown in Fig. 3, the binding list (13) will be extended with bindings of concrete types for the variables ?t1, ?t2, ?t3,?t4 and ?t5. Following that process, all of the difference expressions (8-11) will be processed, resulting in a set of final bindings, for example:

```
((?fptrinst (● bp rsparms9)) (?inst02 (● bp rsparms9)) (?fldname bp)
 (?pptrinst (& b)) (?inst01 b) (?startinst b) (?fldinst (bp rsparms9 :dsvalue (& b)))
 (?struct rsparms9)   (?itype bwimage) (?pscope scopep) (?fscope scopef)
 (?structname rsparms9)
 (?t3 |(dsdeffieldop_field1_(?fldname_rsparms9_:dsvalue_ ?pptrinst_:myscope
        _scopef_:instname_'?fldinst)_ptr2)|)
 (?t4  |(dsdefptr_ptr2_bwi8_:myscope_scopef_:instname_'?fptrinst)|)
 (?t5  |(bwimage_bwi8_:myscope_scopef)|)
 (?t2  |(dsdefptr_ptr1_bwi7_:myscope_scopep_:instname_'?pptrinst)|)
 (?t1  |(bwimage_bwi7_:myscope_scopep_:instname_'?startinst)|)
… …)                                                                    (14)
```

The chain of instances produced by this process is shown in expression (15):

b, (& b), (bp rsparms9 :dsvalue (& b)), (● bp rsparms9), (* (● bp rsparms9)) (15)

Thus, "b" from expression (1) in scopep will map into the expression "(* (● bp rsparms9))" within expression (2), which is in scopef. At code generation time, when the C language surface syntax is added to the internal form, it will be re-expressed as the C language form "(* (rsparms9.bp))".

3 Recursive Type Constructors

The machinery used to synchronize payloads and frameworks uses the CQ Type system to simultaneously define a conventional data type and a design context point of view for that conventional data type. It accomplishes this by defining types as recursive expressions of other types that capture both the data type and the design point of view within some recursive design space. The pattern of types and subtypes will recapitulate the pattern of recursion. That is, a type is some functional composition of its subtypes. For example, a pointer to a BWImage type is a subtype of a BWImage. Thus, type/subtype structures mimic the specialization and generalization of a design space. However, not all design space relations are specialization or generalizations. Some relations are bridges between design spaces that neither specialize nor generalize design context. They transform design contexts. That is, they establish a transformation or a mapping between design contexts. The cross connection transformation (Ref. 3-09) is just such a bridge. In this case, the design point of view for the BWImage entity has transitioned from a simple design context (i.e., the payload) to a different design context (i.e., the framework context containing the “glue” code manufactured by the generator to synchronize the computational contexts). Within the framework context, the BWImage data entity must have the computational form of a value within a field of the struct manufactured by the generator. That is, in the payload context, the BWImage entity might have the computational form “b” whereas in the related framework context, that data entity might have the related computational form “(*(csparm9.bp))”. The computational forms both refer to the same entity but from different design context points of view.

In general, transitions within a design space, be they generalizations, specializations or bridge transformations, represent a transition of design point of view. And this transition may or may not require a transformation of the computational form for a data entity. So, how do we build such a design space?

The system that implements this machinery is defined by a few basic type constructor elements. A sampling of these constructor elements is expressed below in extended BNF, where non-terminals are enclosed in angle brackets and terminals are enclosed in double quotes. Square brackets indicate optionality, braces indicate grouping, and double bars indicate alternation. Parentheses indicate CommonLisp like list and sublist structures that express the form of an instance. Finally, “::=” means “is defined as.”

The following sampling of definitions illustrates the form of some typical CQ Type constructor expressions and includes examples, some drawn from expression 14:

<ArrayType**>::=**(“DSDefArray”[<tag>](<D1><D2>...)<atype> <keyword parms>)
<u>Example CQ Type</u>: |*(dsdefarray_array4_(m_n)_colorimage_:instname_'?foo)*|

<FieldType**> ::=** (“DSDefFieldOP” [<ftag>]
(<fieldname> <structtag> <keyword parms>)
<resulttype>)

Example CQ Type: |*(dsdeffieldop_field1_(?fldname_rsparms9_:dsvalue _ ?pptrinst_:myscope_scopef_:instname_'?fldinst)_ptr2)*|

<StructType**> ::=** ("DSDefStruct" [<stag>]
((<ptype1> <name1> <keyword parms>)
(<ptype2> <name2> <keyword parms>)...))

<PointerType**> ::=** ("DSDefPtr" [<ptag>] <supertype> <keyword parms>)
Example CQ Type: |*(dsdefptr_ptr1_bwi7_:myscope_scopep_:instname_'?pptrinst)*|

<C-type-specs**> ::= {** [<storageclass>] || [<typequalifiers>] || [<type-spec>] }
<PLType**> ::=** (<type> [<ttag>] [<C-type-specs>] [<keyword parms>)])
Example CQ Type: |*(bwimage_bwi7_:myscope_scopep_:instname_'?startinst)*|

Beyond the CommonLisp syntactic framework in which these type specifications are embedded, these CQ type expressions also borrow from the C programming language, which is the default language emitted by the generation system. As with their C counterparts, the optional "tag" fields provide a handle for the composite type. The <keyword parms> (e.g., the keyword ":myscope" paired with the value "ScopeF") provide the mechanism for qualifying types with design features, design contexts or other qualifications specific to the generation process. Keyword based qualifications are "meta-qualifications," which is to say that they are "meta" to data types that are specific to the *programming language domain*. The list of <ptype *n*> groups within StructType definition is shorthand for a list of **<**FieldType**>** types.

In addition to program generation qualifiers, the qualifiers used in the <PLType> definition may also include vanilla programming language data types and type qualifiers, e.g., C type qualifiers from <C-type-specs>. For example, <storageclass> may include auto, extern, register, static and typedef; <typequalifiers> may include const, restrict, and volatile; and <type-spec> may include short, long, signed, and unsigned.

Declarations are defined using a CQ type specification (e.g., <CQ Type Expression>) in place of a simple programming language data type, for example:

<Declaration> **::=** (DSDeclare <CQ Type Expression> [<Instance>])

4 Related Research

From the most general perspective, the purpose of this research is program generation. For general background, see [1-5] and for the DSLGen umbrella research context of this paper, see [6]. However, the details of the machinery in this paper are most closely related Programming Data Type research. Therefore, this section will focus largely on the relationships and differences between the CQ Type research area the programming date type research area. (See [8-9] and [11-12]). Broadly speaking, data types have been used in the pursuit of several different but related objectives:

- **Type checking and inference** to reduce errors in the context of strongly typed programming languages,
- Program language design for **enhancing reusability of code** (e.g., through abstract data types, object oriented programming and polymorphism),
- Program **language design** to simplify programming and enhance the ability to write correct programs (e.g., via functional and applicative languages where types may play a significant role in specification and compilation),
- **Writing correct programs** from formal specifications (e.g., stepwise refinement),
- **Formal specification** of the "meaning" (i.e., semantics) of computer programming elements (e.g., denotational semantics and models like the Z language or the VDM method).

The main difference between the CQ Types research and previous data type research is that CQ Types research is operating in the *design domain space* whereas previous work was to a greater or less extent operating in the *programming language domain space*. That is, previous conceptions of types did not provide a way to express design concepts except to the degree that those design concepts were expressible in terms of programming language constructs or abstractions thereof.

To choose a concrete example, consider the Liskov substitution principle. This principle "seems" to bear some cosmetic relationship to CQ Types and their associated operations in the sense that the work is characterizing the situation in which a subtype is computationally substitutable for its super type in a piece of code. That is, the Liskov substitution principle is superficially similar to the CQ Type work in the sense it deals with mapping one form of programming language code to a related form. However, it is different it two obvious and important ways. The Liskov substitution principle works strictly within the domain of programming types and the type/subtype relationship depends on an *implicit* property of both. CQ Types are defined specifically to work within and capture knowledge about the design space (as opposed to the programming language space) and thereby involve entities (e.g., design features and design contexts) that are not explicit elements of the programming language domain. Furthermore, the relationships among CQ Types are *explicit* rather than implicit and they explicitly capture differences in computational structure that are due to implicit relationships within the design and generation domains (i.e., outside of programming language domain).

5 Summary and Conclusions

CQ Types provide a mechanism for expressing abstract relationships between entities within a conceptual design space and thereby for specifying plans for adapting and relocating code from one design context to another. What constitutes a context is completely open. A context might represent a simple design feature, a locale or scope, a set of computational states, an abstract design for code, a computational partition, etc. This expressive freedom allows the generator to evolve logical designs for a

computation by adding elective design features (e.g., threaded parallelism) to pedestrian and perhaps inefficient designs of a computation while still having the capability to automatically convert from one design form to another.

Furthermore, CQ Types and their associated type difference transformations can be abstracted to precursor equations that parameterize the concrete entities and relationships, which will become concrete via instantiation from some future concrete code within some future payload instance. This allows design frameworks to factor out and express only those design elements and relationships that are determined by the framework, e.g., the relationship between a yet-to-be-determined variable in a payload and its yet-to-be-determined field representation within a structure that connects the payload data to the framework operations. Once such a concrete future payload has been identified to the generator, the yet-to-be-determined elements can be computed by the generator to complete a fully integrated design for the target program.

References

1. Ted J. Biggerstaff, "A perspective of generative reuse, Annals of Software Engineering," Baltzer Science Publishers, AE Bussum, The Netherlands, 1998, pp.169-226.
2. Ted J. Biggerstaff, "A new architecture of transformation-based generators," IEEE Transactions on Software Engineering, Vol. 30, No. 12, Dec., 2004, 1036-1054.
3. Ted J. Biggerstaff, "Automated partitioning of a computation for parallel or other high capability architecture," Patent no. 8,060,857, United States Patent and Trademark Office, filed January 31, 2009, issued November 15, 2011.
4. Ted J. Biggerstaff, "Non-localized constraints for automated program generation," United States Patent and Trademark Office, Patent no. 8,225,277, filed April 25, 2010, issued July 17, 2012.
5. Ted J. Biggerstaff, "Synthetic partitioning for imposing implementation design patterns onto logical architectures of computations," United States Patent and Trademark Office, Patent no. 8,327,321, filed August 27, 2011, issued Dec. 4, 2012.
6. Ted J. Biggerstaff, "Reuse: Right Idea, Wrong Representation?" Invited paper in DReMeR 13 – International Workshop on Designing Reusable Components and Measuring Reusability, Pisa, Italy, June 18, 2013.
7. Ted J. Biggerstaff, "Automated Synchronization of Design Features in Disparate Code Components Using Type Differencing," United States Patent and Trademark Office, Patent no. 8,713,515, issued April 29, 2014.
8. Luca Cardelli and Peter Wegner, "On Understanding Types, Data Abstractions, and Polymorphism," Computing Surveys, Vol. 17, No. 4, December, 1985.
9. Luca Cardelli, "Type Systems," CRC Handbook of Computer Science and Engineering, 2nd Edition, Ch. 97, 2004.
10. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns, Addison-Wesley, 1995.
11. Barbara Liskov and Jeannette Wing, "A Behavioral Notion of Subtyping," ACM Transactions on Programming Languages and Systems, 1994.
12. Barbara Liskov and Jeannette Wing, "A Behavioral Subtyping Using Invariants and Constraints," Carnegie Mellon Report CMU-CS-99-156, 1999.
13. Timothy G. Mattson, Beverly A. Sanders and Berna L. Massingill, Patterns for Parallel Programming, Addison Wesley, 2008.