Synthesis of Concurrent Garbage Collectors

Douglas R. Smith
Stephen J. Westfold

with Cordell Green, Christoph Kreitz, Jim McDonald, Peter Pepper, Eric Smith, Edwin Westbrook

and Alessandro Coglio, Cordell Green, Eric Smith on Flex seedling project

Kestrel Institute

15 October 2015

Contract Number: FA8750-10-C-0241
Effective Date: 23Sep10
Expiration Date: 23Sep15

Principal Investigators: Douglas R. Smith, Stephen J. Westfold

Prepared for:

Air Force Research Lab/RIOIP
26 Electronic Parkway
Rome, NY 13441-4515

Table of Contents

Table of Figures

1 SUMMARY

There were two efforts funded under this contract. The main effort, called CGC herein, focused on the automated generation of secure and correct-by-construction concurrent garbage collectors, and was funded the DARPA I20 CRASH program. A related seedling effort, called Flex herein, focused on the notion of a self-improving theorem-prover.

Kernel functions in a system have a privileged position and can be a source of security vulnerabilities. The challenge of this project was to take a clean-slate design approach to explore ways to produce kernel functions together with proofs of their safety and security. Our focus was on garbage collection algorithms that work both sequentially (stop-the-world collectors) as well as concurrently with an application. Our approach is based on formal specifications of safety and security properties, automated refinement to transform high-level specifications down to code, and the emission of proofs during the development process. The technologies that we developed are applicable to broad range of problems, and they are being applied and further developed in DARPA HACMS and other programs.

We briefly list the highlights of the research and development performed under this contract.

(1) Mixed Logical/Algebraic/Coalgebraic Specifications — Specware naturally supports the introduction of underspecified types and functions, which is necessary for an automated refinement approach. We found a natural way to specify coinductive types and their operators and to refine them to imperative code. Support for both inductive and coinductive types has enabled a far more flexible specification and refinement language, and has supported the generation of true system code.

(2) New Transformations — We developed, implemented, and extensively used a suite of new transformations that generate correct-by-construction refinements.

- Observer Maintenance — This transformation takes an observer and an invariant that characterizes its meaning. It calculates updates codes for each transformer that serve to enforce the invariant.

- Observer Refinement — This transformation takes an observer whose observation type is abstract and provides a more concrete implementation of the type.

- finalizeCoinductiveType — This transformation and its dual (finalizeInductiveType) allow us to incrementally add observers (resp. constructors) to a type. This supports a refinement process where we incrementally add constraints to types and their operators. The transformation generates a definition for the type and gives definitions for functions that are constrained by coinductive (resp. inductive) axioms.

- Globalize — The coalgebraic style of specification uses linear, or single-threaded, functions to express the dynamics of a state-changing dynamical system (such as a concurrent GC). The state of the computation is both an input parameter and a single output parameter. The single-threadedness allows this transformation to suppress the state parameter and treat it instead as a global variable. From purely functional specifications, we are then able to generate idiomatic imperative code.

(3) Proof Emission from Transformations — We pioneered techniques for extending our transformations so that they not only generate more concrete specifications, but they simultaneously output a checkable proof that the refinement is correct. Generating proof scripts as a by-product of each refinement step is only possible because our transformations operate by performing explicit calculations in the domain theory (i.e. using domain axioms and theorems). We generate proof scripts expressed in the Isar sublanguage of Isabelle, which can be checked automatically. For our Mark&Sweep collectors we automatically generated over 33,000 lines of proof text, which was then proof-checked by Isabelle.

(4) Derivations of Garbage Collectors — We developed metaprograms that automatically generate a family tree of the most common classes of garbage collectors from a common specification. For experimental purposes, we wrote a program that randomly generates garbage.

- Mark and Sweep concurrent collectors — We generated a series of increasingly efficient versions of Mark&Sweep collectors, both state-the-world sequential collectors and concurrent collectors.

- Copying Collectors — We generated a version of the Cheney Copying Collector. A key technical innovation was recognizing that the key safety property — the collector must ensure that the live graph remains isomorphic under its actions — provides a key driver of the derivation process. The necessity to enforce isomorphism of the heap allows us to calculate the copying steps of the Cheney Algorithm. In other words, we can replace invention by calculation.

- Generational Collectors — We generated a generational collector by means of a relatively easy modification to the metaprogram for generating a Cheney Copying collector.

- Reference Counting — We generated a series of reference count collectors, where the key insight was to maintain the reference count as an invariant. This allowed the Observer Maintenance transformation to do most of the work of calculating updates to the reference count for each node and to trigger recycling of dead nodes.

(5) Generator of Imperative & Concurrent Code — We extended our code generators to allow the coalgebraic specifications to be translated to idiomatic state-changing CommonLisp code. Efforts to generate idiomatic C code were underway as the project concluded.

(6) Demonstrated Software Evolution via Metaprogram Evolution — One of the surprising results of this project was the degree of commonality between the metaprograms for the different collector algorithms listed above. When starting a derivation for a new class of collectors (essentially in the order listed above), we started with the derivation metaprogram and modified it, rather than starting from scratch for each. We found that over 65% of the metaprogram text survived verbatim.

Flex Seedling

We have investigated the idea of building *Flex*, a self-improving theorem prover. The vision is that of an automated theorem prover that can apply transformations (optimizations) to itself, timing itself on a test suite and attempting to find sequences of transformations that improve its performance. Flex adds automation and self-adaptation to the automated refinement technology used in the garbage collection synthesis effort.

We have built a prototype automated theorem prover and demonstrated how various transformations lead to performance improvements. We have built prototype functionality to automatically generate a potentially large number of transformed versions of the prover and to run a set of tests on it. While much remains to be done to fulfill the Flex vision, the results we obtained under this seedling are promising.

2 INTRODUCTION

2.1 Motivation

The DARPA CRASH program sought to develop a clean-slate approach to producing secure host computers, loosely following a biologically-inspired approach. It addressed the broad question: If we could design a host computer with security as a key requirement, including all of its hardware and software layers, how would we proceed?

Kestrel proposed to focus on the correct-by-construction generation of a key component of the language runtime of many systems: its runtime memory management or garbage collection algorithms. These algorithms usually run in the OS kernel and have privileged access to data. They have been exploited to leak sensitive information (violate implicit confidentiality requirements), and can potentially be used to breach integrity and availability requirements.

A key part of our effort was (1) to generate safe, secure, and performant collectors, and (2) to generate proofs as a by-product of the synthesis process, so that certifying authorities could independently and efficiently check that the collectors satisfy their safety and security properties. In effect we aimed to generate proof-carrying programs.

Taking a larger view, the correct-by-construction generation of garbage collectors can be seen as an instance of (paves the way toward) the ability to generate critical software components in the software stack, together with certification evidence. The technology is applicable to a broad range of problems, and its being applied and further developed in DARPA HACMS and other programs.

2.2 Objective & Hypothesis

A long-term goal has been to demonstrate that the automated generation and evolution of software from requirements-level specifications provides a cost-effective alternative (or supplement to) current methodologies for software development. Benefits of the approach include correctness-by-construction, generation of certification evidence in the form of proofs, good performance, and productivity gains through automation. Our specific objective in this project has been to demonstrate the feasibility of automating the generation of a family of concurrent garbage collectors. In Section 4 we discuss the extent to which our results advance our long-term goal and demonstrate the claimed benefits.

2.3 What is program synthesis and what is our approach?

Generally, program synthesis is the automated construction of programs from specifications of their intended behavior. Our approach is deductive and starts with the capture of program/system requirements in terms of formal specifications. The full power of higher-order classical logic, as supported in Kestrel's Specware system [SW03], is used to express specifications as first-class entities along with operations for structuring, composing and refining them.

A major distinguishing feature of our refinement approach is that we can calculate each of the refinement steps automatically. Other approaches to refinement (e.g. VDM, B, Praxis) rely on the post-hoc verification of manually created refinement steps. This is an expensive process, and it proves difficult to maintain the refinement chain under changes in requirements. While there is a significant upfront investment in building up a domain-specific specification for garbage collection (or other domains), the payoff comes downstream with the automated generation of families of codes together with their proofs. The amortized cost over the product family and over its lifecycle should be dramatically lower than for other approaches to software production.

The synthesis approach that we developed in this project requires user input in two parts: (1) formal requirements specification, and (2) a metaprogram.

The development of correct-by-construction code via a formal refinement process has the following form:

$$Spec_0 \Leftarrow Spec_1 \Leftarrow \ldots Spec_n \Leftarrow Code.$$

The refinement process starts with a specification $Spec_0$ of the requirements on a desired software artifact. Each $Spec_i$, for i=0,1,...,n represents a structured specification and the arrows $\Leftarrow$ are refinements. The refinement from $Spec_i$ to $Spec_{i+1}$ embodies a design decision which narrows down the number of possible implementations. The final step translates the lowest-level specification $Spec_n$ to code in a suitable programming language. Semantically the effect is to narrow down the set of possible implementations of $Spec_n$ to just one, so specification refinement can be viewed as a constructive process for proving the existence of an implementation of specification $Spec_0$; i.e. its consistency.

2.4 What is Garbage Collection?

Many modern programming languages provide support for dynamically allocated memory. In contrast to local variables in a function which can be stack allocated since their lifetime is know statically to end when the function returns, dynamic memory is allocated from the heap since its lifetime is not, in general, knowable at compile-time. This entails the need for a runtime component that tracks objects on the heap to decide when they are no longer needed and their memory can be recycled for other purposes. This process is called *garbage collection* and the component is called a *garbage collector*, or simply a *collector*.

To work properly, a collector must have access to any part of a computer that can hold references (pointers) to the heap, including registers, runtime stacks, and the heap itself. This privileged access makes it a potential source of security vulnerabilities, since an attacker that gains control of a collector could access sensitive information, corrupt the state of an application, or tie up space and time resources to degrade the services that depend on the collector.

Similar concerns can be stated many components of the operating system kernel. By focusing on the generation of secure and correct-by-construction garbage collectors, this project aimed to demonstrate a cost-effective way to produce high assurance components in general.

2.5 Flex Seedling

2.5.1 Adaptation

A software system can be more resilient and performant if it can automatically adapt its own behavior to changing external conditions. The adaptation should take place dynamically, while the system is running, as soon as the environmental changes are detected.

Currently, automatic dynamic adaptation is typically limited to a few pre-programmed parameters that can change in response to external stimuli. A major motivation behind Flex is to go significantly beyond that. We envision software that can change its own code by applying transformations to itself, where the new code is optimized to the new environmental conditions.

Scalable: Proofs increase scalability. We assume bugs and their effects are major limiting factors in the scalability of applications. So reducing errors and their effects, via proofs, allows growth to larger scale.

Safe Emergent Behavior: Because we have proofs that constrain/limit the behavior of the software, we can safely let the system evolve, with little risk of undesired emergent

behavior. Of course, proofs help avoid not only “emergent” but also many kinds of undesired behavior.

2.5.2 Underlying Transformation Technology

Our Flex prototype is based on the same formal specification and automated refinement technology described earlier, used for the synthesis of garbage collectors. The starting point for adaptation is a full derivation of an implementation from a specification. Adaptation is achieved, in the Flex vision, by automatically modifying the derivation based on the environmental changes to adapt the system to, and by re-generating a new implementation that is optimized to the new situation.

In order to have tighter integration between derivations and proofs, we carried out most of our Flex development in a Specware-like extension of the industrial-strength ACL2 theorem prover [ACL2]. The Specware-like extensions was developed under a separate effort: it features specifications and morphisms as described earlier, as well as a collection of automated proof-emitting transformations.

2.5.3 Resolution Theorem Proving

Flex is a theorem prover based on resolution [Robinson65], a proof procedure that underlies some of the world’s best theorem provers. The use of a resolution theorem prover for program synthesis was pioneered in [Green69].

A resolution theorem prover works by refutation: to prove that a conclusion C follows from a set of hypotheses H, the theorem prover attempts to derive a contradiction from H and the negation of C. Both H and the negation of C are supplied in conjunctive normal form, i.e. as a conjunction of clauses, where a clause is a disjunction of literals, where a literal is either an atomic formula or the negation of one. For more information on resolution and related theorem proving techniques discussed in this report, see [Wos00].

3 METHODS, ASSUMPTIONS, PROCEDURES

Our objective was to demonstrate the feasibility of automating the generation of a family of concurrent garbage collectors from requirement-level specifications, together with correctness proofs. The overall approach is characterized by formal specifications, formal refinements, transformations to generate refinements, proof-emitting transformations, and metaprograms. A key assumption is that it will ultimately become practical for system developers to capture their requirements as formal specifications, and that the process of transforming those specifications to proof-carrying code can be largely automated (with some guidance).

3.1 General Approach

The synthesis approach that we developed in this project requires two forms of user input: (1) formal requirements specification, and (2) a metaprogram.

metaprogram =
transformation_1 ;
transformation_2 ;
transformation_3 ;
…

Figure 1: Form of a Metaprogram

Figure 1 shows a metaprogram as a sequence of transformations. The metaprogram is a sequence of transformations to be applied to the requirement specification. The transformations are typically drawn from Specware's library. In a later section, we present a collection of new transformations that were developed as part of this project. The actual syntax/representation includes parameters to the transformations as well any theorems that should be applied.

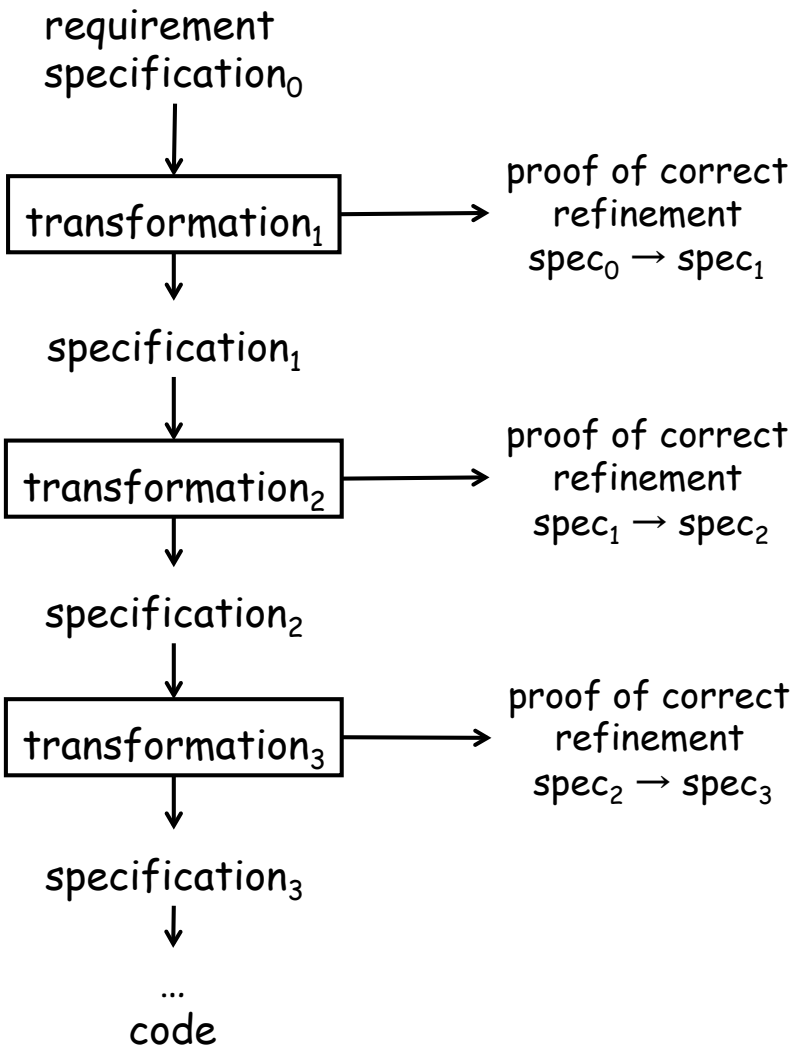


Figure 2: Executing a Metaprogram to generate code and proofs

Specware executes a metaprogram automatically. The effect, illustrated in Figure 2, is to sequentially transform the requirement specification into more refined specifications. Each transformation embodies some design knowledge, so the effect of applying a transformation is to generate (1) a refinement of the input specification into a refined specification that incorporates an instance of the transformation's design knowledge, and (2) a machine-checkable proof that the output specification is a refinement of the input specification. The metaprogram is then an explicit and formal statement of the design content of the generated code.

We would like to emphasize the consequences of this approach with respect to software evolution. Studies of evolution suggest that most changes to systems fall into a small number of categories, mainly bug-fixes, additionally requirements, performance tuning, and migration. First, bug fixes are not relevant here since the code is generated with

proofs of correctness (although bug fixes to the requirement specification will commonly arise). Second, the addition of requirements is facilitated in our approach because we have a formal specification of requirements. It is much easier to add requirements to a specification than to add them at the code level. Third, performance tuning also has an explicit locus in our approach, since it is manifest by extending or modifying the metaprogram — either adding new transformations, or modifying how existing transformations are applied (e.g. by adding theorems that the transformation can use). Finally, migration is often a matter of adapting the metaprogram to suit a new target language or platform. Typically most of the metaprogram is preserved under migration with just some of the backend transformations needing to be changed. In summary, our approach, based on formal specification of requirements and the derivation structure via a metaprogram, provides good locality for the kinds of changes that arise in software evolution. This fact underlies our claim that this approach is essential to the future of Software Engineering.

3.2 Specifications and Refinement

A *specification* defines a language and constrains its possible meanings via axioms. A specification is given by a finite collection of type symbols (optionally including a definition), function symbols and their signature (optionally including a definition), and axioms over the type and function symbols. We treat predicates as Boolean-valued functions. For purposes of this paper, we focus on first-order specifications (i.e. functions do not take functions as arguments), although Specware allows higher-order specifications. The deductive closure of the axioms is a theory, so a specification is a finite presentation of a theory.

A refinement can be expressed formally via a *specification morphism* which translates the language of one specification into the language of another specification in a way that preserves theorems. Formally, a *signature morphism* from specification S0 to specification S1 is a type-consistent map from the vocabulary of S0 (i.e. its type and function symbols) to the vocabulary of S1. A *specification morphism* from S0 to S1 is a signature morphism that preserves theorems; i.e. that translates each theorem of S0 to a theorem of S1. To establish a specification morphism, it is sufficient to prove that each axiom of S0 translates to a theorem of S1. Let Morphism denote the type of specification morphisms (or simply morphisms).

Specification S1 is an *extension* of specification S0 if there is an specification morphism S0 → S1 whose underlying signature morphism is injective. We use importation to express extension, allowing the construction of complex specifications. More generally, specifications and their morphisms constitute a co-complete category, where the colimit operation provides a general means for constructing complex specifications. Intuitively,

the colimit is the simplest specification that combines given specifications C modulo their common structure.

As models of specification S, we admit any structure of sets and functions that interprets at least of each type and function symbol in S and that satisfies the function signatures and the axioms. This semantics allows structures for extensions of S to be models of S. The denotation of a specification morphism m is a map from models of the codomain of m into models of the domain — every model of S1 is mapped to a model of S0.

3.3 Proof-Emitting Transformations

Specification S0 *refines to* S1 if there is a specification morphism m:S0 → S1. We refer to m as a refinement and a morphism, and in context, S1 as the refinement of S0. In this paper we are interested in rules and techniques for automatically generating refinements. A s*pecification transformation* or simply a *transformation*, is a partial function on specifications that generates a refinement: t:Spec → Morphism. That is, if t(S) = m, then m:S→ codomain(m) is a refinement of S.

As discussed in the next section, we developed a set of new transformations that support a coalgebraic style of specification, leading towards the generation of imperative and concurrent code. Most of our transformations work by applying a sequence of equations (via rewrite rule) to parts of the given specification. The chain of equations that are applied proves the correctness of the resulting refinement. We developed techniques for saving the equation chain and emitting it as a proof structure that can be checked by an external proof checker. In our case, the proofs are expressed in the Isar format of the Isabelle proof assistant. Isabelle is used to automatically check that the emitted Isar proofs are in fact proofs of the refinement proof obligations generated by Specware.

The upshot of using proof-emitting transformations is to co-generate both code and proof that the code satisfies its specification. This is in contrast to post-hoc verification approaches that seek to prove a program correct after it has been written. Generating proof-carrying code has the advantage that all design information is available to the proof generation process as the code is being constructed. We believe that this is a more economical approach to producing certifiably correct software.

3.4 Coalgebraic Specifications

Coalgebra is a relatively recent area of mathematical study, which, in a sense, is dual to algebra. It has been attractive as a way to model and reason about infinite and non-well-founded objects, such as streams and the behaviors of state machines [Rutten00,Jacobs97]. It has been a natural unifying foundation for exploring dynamical systems, including both discrete and hybrid automata. Coalgebra also provides a natural way to model classes in an Object-Oriented sense and subclass hierarchies.

For our purposes, algebra, via inductive types, provides a foundation for specifying and refining finite data, such as Booleans, Natural numbers, Lists, and finite Sets. Coalgebra, via coinductive types (aka cotypes), provides a natural foundation for specifying and refining stateful and concurrent computation. It has also proved useful for giving a foundation to object-oriented languages and class hierarchies [Jacobs97].

One of our new approaches in CRASH was to use a mixture of algebraic and coalgebraic types in our specifications, and to develop new transformations to handle the cotypes.

There is a descriptive vocabulary that goes with cotypes. Algebraic types are characterized by their constructors, which are used inductively to build up terms for all values in the type. The inductive construction allows inductive definitions of functions and proof by induction. In contrast, cotypes are characterized by their destructors, which are operations on the cotype that decompose a cotype element into its parts. Typically destructors are categorized as *observers* (which observe an aspect of an element) or *transformers* (which transform an element into another element of the cotype). The iterated destruction of objects of the type give rise to coinductive definition of functions and proofs by coinduction.

Here is a generic specification that illustrates the coalgebraic style that we developed in this project:

```
S = spec
  cotype State
  op obsA:State-> A
  op obsB:State-> B
  op obsC(st:State):C = h(obsA st, obsB st)
  op f(st:State)(arg:Arg):
       {st':State| obsA st' = alpha obsA st
                  & obsB st' = beta  obsB st}
  op g(st:State)(arg:Arg):
        {st':State,d:D| obsA st' = gamma (obsA st) arg
```

```
                    & obsB st' = delta (obsB st) arg
                       &       d = eps (obsA st) (obsB st) arg}
    end spec
```

Spec S has two basic (undefined) observers (obsA and obsB), a defined observer (obsC), and two transformers/destructors (f and g). The latter are specified by giving coinductive constraints (postconditions) stated as the predicates of a dependent output type. That is, the output type of f is the set of all States st' such that the obsA observation is given by (alpha (obsA st)); i.e. by some function of the old obsA observation. This is a dependent type because it depends on the value of st that is bound when the transformer is called. The types A, B, and C may be algebraic (i.e. constructor-based). The Greek-letter functions (alpha, ...) capture the effect of the transformer on their particular observer.

Here is a simple specification of mutable graphs using this style:

```
Graph = spec
  cotype Graph
  op nodes : Graph -> Set Node
  op sucs  : Graph -> Node -> Bag Node
  op addArc (G:Graph)(x:Node,y:Node):
        {G':Graph |  nodes G' = nodes G
                  & sucs G' x = insert(y, (sucs G x))}
  end-spec
```

Spec Graph introduces an undefined cotype Graph that has two observers, nodes and sucs, and one transformer addArc. All that we know about a Graph is what we can observe, which is its current set of nodes and the successors of any given node. The addArc transformer allows us to change a Graph by adding a new arc from node x to node y. The style of specifying the addArc operation is via predicates expressed in coinductive form: the result of adding an arc is completely specified in terms of what observations we can make of the new Graph.

Here is a more elaborated specification of mutable graphs that is closer to the form that we ultimately settled on in the GC derivations.

```
Graph = spec
  cotype Graph
  type NodeId          % identifiers of Nodes
  type Index
  op roots: Graph -> Set NodeId
  op allindicies: Graph -> NodeId -> Set(Index)
```

```
  type Arc = Map(Index, NodeId)
  op src(G:Graph)(n:NodeId)
  op tgt (G:Graph) (a:Arc):NodeId
  op nodes : Graph -> Set NodeId
  op outNodes  : Graph -> NodeId -> Set NodeId
  op setTgt (G:Graph)(a:Arc)(y:Node):      % swing the arc a to point to y
          {G':Graph |  nodes G' = nodes G
                       &  roots G' = roots G
                       &    tgt G' a = y }
end-spec
```

A stateful setting allows values of ``variables'' to vary with changing state. In coalgebraic terms, observations of state will vary over time. This gives rise to the key distinction between identity and value: over time the identity of an observation remains stable (is preserved) while its value may vary. This phenomenon is pervasive in everyday life as well as in formal contexts; e.g. citizens have a unique identifier for government purposes (e.g. their SSN) while the value of the citizen's age and weight, say, varies over time. Similarly in a formal context, an IP address provides a unique identifier for Internet purposes, but it refers to (its value may be) a constantly varying local network.

In this style of specification, it is important to begin formalization with an understanding of what the observers are, and the distinction of identity versus value. For a single fluent (changing value), a simply observer of state is sufficient. For a (more or less) structured collection of values, an observer that is parametric both on state and unique identifiers for the values is needed. That is, the observer function itself is a unique identifier, but if there is a collection of changing values, then an identifier type Id must be introduced and an observer that is parametric on Id is introduced to observe individuals of the collection.

The behaviors of this system would again be all streams of Graph, induced by newGraph and the addArc and setTgt transformers (transition functions). Other observers and transformers will be added as needed.

For example, in a formal specification of mutable graphs, we have the nodes and arcs as observable entities. A specification is required then to have types NodeId and ArcId, together with observers that are parametric on those identifier types to yield the current values of nodes and arcs.

The transformers are only known via the changes that they make to observations, leading to a coinductive style of specification, expressed by coinductive constraints in the postcondition of transformers.

3.5 Specification of Concurrent Garbage Collectors

The domain specification of a collector is built up incrementally. In the previous section we presented a fragment of a generic specification for mutable/coinductive graphs. We now extend Graph to Heap by adding heap concepts: nodes can be roots, and be live, dead, supply, and active.

```
Heap = spec
  import translate Graph by {Graph +-> Heap}
  op roots  : Heap -> Set NodeId
  op supply : Heap -> Set NodeId
  op active(H:Heap) : Set NodeId =
      lfp( roots H,  fn(ns:Set NodeId)-> (allOutNodes H ns))

  op live (H:Heap): Set NodeId =  active H \/ supply H
  op dead (H:Heap): Set NodeId =  nodes(H) — live(H)
end-spec
```

Heap imports the Graph specification and, in the process, renames the cotype Graph to Heap. It introduces new observers of the Heap: the roots, active nodes, live nodes, and dead nodes. The *live* nodes are the set of nodes that can be reached from registers, the stack, and static memory via references. We omit axioms asserting that all of these observe subsets of the current Heap's nodes. [1]

Next, we further extend the Heap specification to Collector by adding new observers and transformers relevant to collectors.

```
Collector = spec
  import Heap
  op black(H:Heap): {blk:Set NodeId | blk subset (nodes H)}

  op insertBlack(H: Heap)(n:NodeId | n in? nodes H)
      : {H': Heap |  black  H' = set_insert(n, black H)}

  op deleteBlack(H: Heap)(n:NodeId| n in? black H && n in? nodes H)
      : {H': Heap |  black  H' = set_delete(n, black H)}

  refine def addSupply(H:Heap)
```

[1] Notation: \/ is set union and — is set difference.

```
                    (n:NodeId | ~(n in? black H)
                              && ~(n in? supply H)
                              && n in? nodes H)
      : {H': Heap | black  H' = set_insert(n, black H)}

    op findLive (H:Heap | black H = supply H) : {H':Heap | live H' subset black H' }

    op sweep (H :Heap | live H subset black H):
           {H':Heap | supply H' = (nodes H — black H) \/ (supply H)
                    && black H' = empty_set }

    op recycle1(H:Heap): Heap
         = (let _ = writeLine "GC invoked" in
           let H1 = findLive H in
           let H2 = sweep H1 in
           H2)

    op selectSupply(H:Heap): Heap*Option(NodeId) =
       (if supply H = empty_set
          then (let H1 = recycle1 H in
               if size (supply H1) <= thrashBound H1
                 then let _ = writeLine "memory exhausted!" in
                      let _ = throw_abort () in
                      (H1,None)
               else selectSupply H1)
        else  selectSupply1 H)

     op selectSupply1(H:Heap | ~(supply H = empty_set)):
                 {(H',on):Heap*Option(NodeId) |
                     ex(y:NodeId)( y in? supply H && on = Some y
                                  && supply H' = set_delete(y, supply H)
                                   &&  black H' = set_delete(y, black H)
                 }
end-spec
```

Collector imports the Heap specification. It introduces a key new observer, the black nodes which are a computable approximation to the live nodes, and two transformers for modifying black. The color metaphor comes from Dijkstra [Dijkstra78]. findLive is specified to make the black nodes be a superset of the live nodes. Ideally we should establish black = live, but in a concurrent setting the best that can be done is to

establish black as an upper bound. This implies that the complement of the black nodes are guaranteed to be dead nodes. recycle1 performs one iteration of finding live nodes and then returning known dead nodes to the supply. selectSupply removes a node from supply and returns it.

In a stop-the-world setting, where the mutator is stopped while the collector scans for live nodes, it is relatively easy to determine the dead nodes for recycling. When the collector is intended to run concurrently with the mutator, the situation is trickier. Figure 3 shows the structure of a rely-guarantee specification for a concurrent collector.

The collector relies on (or assumes that) its environment monotonically increases dead nodes; technically, that every state-changing action of the Mutator satisfies the specification

CollectorRelyCond (st;State):{st':State | dead st ⊆ dead st' }

the satisfaction of which is indicated by the cross arc from the Collector's rely condition to the Mutator's guarantee condition.

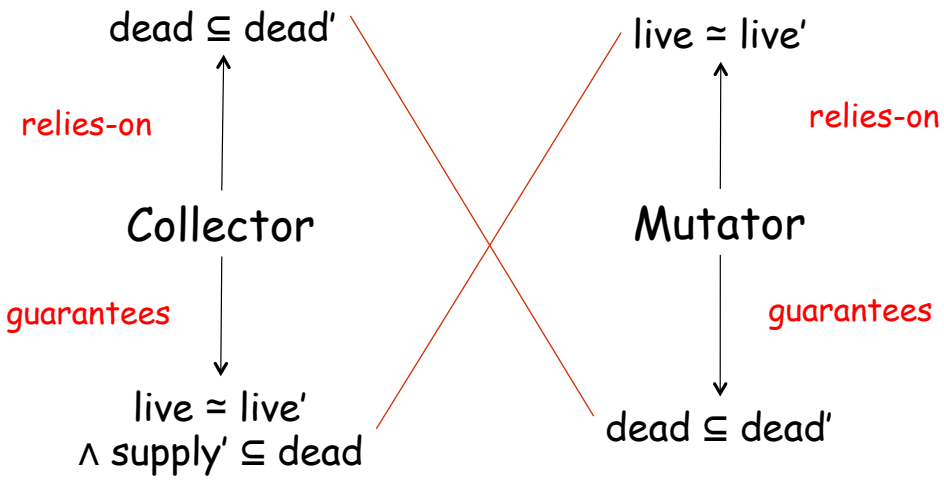


Figure 3: Assume-Guarantee Specification Composition

Conversely, the Mutator relies on (or assumes that) its environment preserves the live graph (via isomorphism); technically, that every state changing action of the Collector satisfies the specification

MutatorRelyCond (st;State):{st':State | live st ≅ live st' }

the satisfaction of which is indicated by the cross arc from the Mutator's rely condition to the Collector's guarantee condition. If the Collector's rely condition is satisfied by its

environment (i.e. the Mutator) then it guarantees each of its actions preserved the live graph (isomorphically) and that the supply nodes in the end state are all dead nodes. Conversely, if the Mutators rely condition is satisfied by its environment (i.e. the Collector) then it guarantees each of its actions monotonically increase the dead nodes.

The safety of the composed Mutator + Collector system can be treated more formally as follows. We define a cotype called State that has various observers, including the heap as a rooted graph, ghost observers for the live and dead nodes, and others.

```
cotype State            % The basic cotype
type Transformer = (State -> State)
type Observer a  = (State -> a)

op Graph                % rooted directed graphs
op heap: Observer Graph
op live: Observer Graph                 % a ghost op: for spec purposes only
op dead: Observer (Set NodeId)          % a ghost op: for spec purposes only
...
```

We also define an equivalence relation on State that abstracts away State observers owned by the Collector. Two states are equivalent if they are observationally equivalent to the Mutator, in particular that their live graphs are isomorphic and that the Mutator behaves equivalently in equivalent states.

```
op stateEquiv infix 20: State -> State -> Bool
axiom stateEquivalence is
  reflexive stateEquiv && transitive stateEquiv && symmetric stateEquiv

op graphIso infix 20: Graph -> Graph -> Bool
axiom graphIsomorphism is
  fa(G,G') (graphIso G G') =
        (ex(f:Bijection(Graph,Graph)) G = f G'
                                    && (inverse f) G' = G
                                    && a in? G = (f a) in? G')

axiom graphIso_in_stateEquiv is
 fa(st1:State,st2:State) st1 stateEquiv st2 => (live st1) graphIso (live st2)
```

It is convenient to define a type of Transformers and (polymorphic) Observers and then define subtypes for Collector and Mutator transformers that have desired properties as

subtype predicates. Two key insights are (1) the states are stable under Collector operations, and (2) mutator actions form a congruence wrt state equivalence. We specify that any Mutator action must (1) achieve nondecreasing dead nodes, and (2) be congruent wrt stateEquiv. We specify that any Collector action must preserve state equivalence.

```
type Mut = {m:Transformer |
                fa(st:State) (dead st)  subset  (dead (m st))
                && fa(st:State,st':State)
                      (st stateEquiv st') => ((m st) stateEquiv (m st'))}

% The state is stable under Collector actions
  type Col = {c:Transformer | fa(st:State) st stateEquiv (c st)}

  op mutator  : Mut
  op collector: Col
```

Based on the specification above, we formulate and prove the essential safety property of a garbage-collecting system: the composed Mutator and Collector simulate the behavior of the Mutator alone (up to blocking, see Appendix).

Theorem (Safety): run(mutator || collector) simulates run(mutator).

Proof: The proof is by coinduction (or induction if we use a constructor for an initial state) using the state equivalence as the (bi)simulation relation. To do so, we consider traces of the atomic steps of the mutator and collector interleaved (see Figure 4). We define a trace (aka trail) of a system S as a sequence of alternating states and atomic actions of S:

$$<st_0, a_0, st_1, a_1, st_2, a_2, \ldots st_i, a_i, \ldots >$$

Given a mutator step m preceded and followed by zero or more collector steps, we must show that there is a trace of the mutator alone that reflects the action of m. The key step is illustrated in the figure below. Let S1 = run(mutator) and S2 = run(mutator||collector). Consider (co)inductively a trace tr1 of S1 that arrives at a state st1 and a trace tr2 of S2 that arrives at an equivalent state st2. If tr2 proceeds with a mutator step m preceded and followed by zero or more collector steps, we must show that there is a trace of S1 that reflects the action of m and results in an equivalent state.

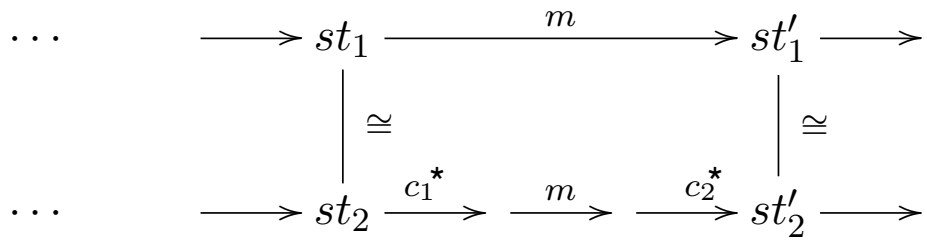


Figure 4: Simulation of Mutator by Collector+Mutator

First, since st1 and st2 are equivalent and m is enabled in st2, then it is also enabled in st1 — this requires that all observations that inform control decisions made by the Mutator are part of the state equivalence. Consequently, there is some extension of the trace prefix of st1 with action m. Next, we show that the resulting states are equivalent:

$$
\begin{array}{lll}
st_2 \cong st_1 & & \text{by assumption} \\
\implies c_1(st_2) \cong st_1 & & \text{by Col subtype property, transitivity} \\
\implies m \circ c_1(st_2) \cong m(st_1) & & \text{by Mut subtype property} \\
\implies c_2 \circ m \circ c_1(st_2) \cong m(st_1) & & \text{by Col subtype property, transitivity} \\
\iff st'_2 \cong st'_1 & & \text{by definition}
\end{array}
$$

where ≅ denotes isomorphism between two graphs. This means that as the composed system S2 simulates S1, step-by-step it preserves isomorphism of the live graph.

3.6 Design Theories and Transformations

In this section, we present a series of novel transformations that generate refinements in our GC derivations. There are three general sources of techniques for generating refinements:

(1) Manual Extensions — manually written extension of a specification
(2) Library Refinements — are applied via a pushout (transformation that invokes colimit computation), and
(3) Transformations — transformations that generate refinements.

As described in Section General Approach, we manually write a metaprogram, also called a derivation script, which is an executable sequence of refinement steps applied to an

initial specification. Each step prescribes how to generate a refinement of the previous specification.

Each of the following subsections introduce a library refinement or a transformation for generating refinements, together with some examples. We also discuss how each technique can automatically generate a formal checkable proof as a byproduct of its action.

Section 3.6.1 introduces an algorithm theory for solving a problem by means of iterating a monotone function to a fixpoint. We apply this to finding the graph of live nodes. Sections 3.6.4, 3.6.5, 3.6.7, and 3.6.8 each introduce a transformation for generating refinements of the observers and transformers of a coinductive type (usually state).

Coalgebraic refinements simply add further constraints to previously introduced transformers, rather than producing constructive definitions. It is only at the end that constructions are given; i.e. that a particular model is chosen. This contrasts with algebraic style refinement in which constructors are given for types and operators are inductively defined over the types. All constructions are explicit and immediate.

3.6.1 Algorithm Design Theory for Fixpoint Iteration

There are two classes of garbage collection algorithms:

- Stop-the-world collectors: these are the classical non-concurrent collectors, where the mutators need to be stopped while the collector works.
- Concurrent collectors: these are the collectors that allow the mutators to keep working concurrently with the collector (except for very short pauses).

The traditional stop-the-world collectors correspond on the abstract level to the classical fixed-point theory of Tarski and Kleene, where the live nodes are the least fixpoint of a monotone function on the graph G. More recently Cai and Paige [CaiPaige89] published a number of generalizations that are streamlined towards practical algorithmic implementations of fixpoint computations. In a concurrent setting the graph G is changing while the Collector is operating, which means (abstractly) that the monotone function itself is changing during the iteration process. In [Pavlovic10] we developed general conditions under which the result of the iteration process is a fixpoint of the initial monotone function but not a least fixpoint. We call this dynamic fixpoint theory, and it justifies the safety of CGCs.

We paraphrase the main result of Cai and Paige here, since it provides the template for our algorithm strategy:

Theorem 1 [Cai-Paige89]

Let A be a cpo[2] and f: A -> A be a monotone function that is inflationary in r. If <s0, s1, s2, ..., sn> is an arbitrary sequence obeying the conditions

$r = s0$,

$si < si+1 \leq f(si)$ for $i < n$,

$sn = f(sn)$

then sn is the least fixed point of f relative to r. Conversely, when the least fixed point is finitely computable, then the sequence will lead to such an sn.

Theorem 1 provides a natural abstraction from workset-based iterative algorithms, which maintain a workset of change items. At each iteration, a change item is selected and used to generate the next element of the iteration sequence. The incremental changes tend to be small and localized, hence this is called the micro-step approach and the Kleene chain the macro-step approach. All practical collectors use a workset that records nodes that await marking.

We now derive the overall structure of a garbage collector. The essence of it is the iterative algorithm for finding a superset of live nodes which we can complement to obtain a subset of dead nodes to recycle. Letting roots denote the roots of the active graph together with the head of the supply list, we have

live = lfp f({})

where

f(R) = roots \Union {b | b∈ G.sucs(a) ∧ ∈ R};

in words, the *active* nodes are the cumulative closure of the roots under the successor function in the current graph G.

To derive an algorithm for computing the *dead* nodes, we simply compute the set of live nodes and subtract them from the set of all nodes, much like in a sieve algorithm. See Figure 5. We can compute the set of live nodes by a correct, but naive iterative algorithm via a Kleene chain; its proof is constructed by instantiating Kleene's proof.

Following Cai and Paige [CaiPaige89], we can construct a more efficient fixpoint iteration algorithm using a workset defined by

WS = f(WSvar) \ WSvar

= roots ∪ {b | b∈ G.sucs(a) ∧ a∈WSvar} \ WSvar

[2] A cpo is a partial order that is complete in the sense that every subset with an upper bound has a least upper bound.

```
findLFP( {} )
op findLFP(S) =
    if S ≠ f(S
    then findLFP( f(S) )
    else S
```

Figure 5: Kleene Fixpoint Algorithm

Although this workset definition is created by instantiating a problem-independent scheme, it has an intuitive meaning: the workset is the set of nodes whose parents have been reached (and ``marked'' as live), but who themselves have not yet been marked.

```
SSP = spec
 type State
 op pre  : State -> Bool
 op post : State -> State -> Bool
 op p (st:State | pre st): {st':State | post st st'}
end-spec

Mealy = spec
 type State
 type In
 type Out
 op pre  : State -> Bool
 op post : State->In->Out->State->Bool
 op p (st:State | pre st)(i:In): {(o,st'):Out*State | post st i o st'}
end-spec
```

Specification SSP} provides an abstract specification of a state-based problem to solve. State} is intended as a coinductive type and p} is specified via a precondition on the input state and a dependently typed postcondition over the input and output states.

A generalization to a Mealy machine is also given: the specified transformer depends both on the initial state and an input, and produces both an output state and an output result.

```
SBFixpointIterationWorksetTheory = spec
 import SSP, StructuredTypes
 op xs: State -> Set X
 type X
 op F : State -> Set X -> Set X
```

```
 axiom F_is_monotone is
   fa(st:State,s1:Set X, s2:Set X)
     s1 subset s2 => (F st s1) subset (F st s2)
axiom fixpoint_solves_p is
     (F st' (xs st')) subset (xs st')  => post st st'
end-spec
```

Specification SBFixpointIterationWorksetTheory provides the structure and sufficient conditions (sufficient structure) to enable a fixpoint solution to a problem given by SSP. This formulation differs from the classical Tarski/Kleene/Paige formulation in that the function F is a monotone function that depends on current state; i.e. in a fixed state it is a monotone function. Our formulation sets the stage for addressing the issue of iterating a monotone function over a changing state due to a concurrent actor.

```
SBFIW_Algorithm = spec
 import SBFixpointIterationWorksetTheory
op WS(st:State): Set X = (F st (xs st)) -- (xs st)
op initialState(st:State): {st':State | xs st' = empty_set}
op nextState(st:State)(x:X): {st':State | xs st' = set_insert(x, xs(st))}
op selectWS (st:State): {(st',ox): State * Option(X) |
                                  if WS st = empty_set
                                  then WS st' = WS st
                                      &&  ox = None
                                  else ex(y:X)(y in? WS st
                                           && WS st' = delete(y, WS st)
                                           && ox = Some y)}

 op p (st:State | pre st): {st':State | (WS st') = {}
                                        && (F st' (xs st')) = (xs st')} =
    let st1 = initialState st in f_iterate st1

 op f_iterate (st: State): {st':State |  WS st' = {} && (F st' (xs st')) = (xs st')} =
   case selectWS st of
     | (st',None) -> st'
     | (st',Some y) -> f_iterate(nextState st' y)

 theorem correctness_of_p is
   fa (st:State,st':State)
      (pre st && st' = p st => post st st')
end-spec
```

A proof of the correctness theorem is as follows: First, we show that f_iterate satisfies its postcondition: at termination, we have selectWS st has returned (st',None), which by the postcondition of selectWS implies that

WS st = WS st' = empty_set,

which further implies that (F st' (xs st')) = (xs st'). Next, we show that p satisfies its postcondition: Assume that p starts in state st and terminates in state st'. Then the postcondition of f_iterate holds in st':

WS st' = {} && (F st' (xs st')) = (xs st')

which is also the postcondition of p. Finally, we show that correctness_of_p is indeed a theorem. Assume that p starts in state st such that pre st, and it terminates in state st'. Then the postcondition of p holds, but then axiom fixpoint_solves_p implies that post st st'.

The algorithm theory above is formally expressed as a morphism

fixpoint: SBFixpointIterationWorksetTheory → SBFIW_Algorithm.

We formulate it, prove it, and store it in the library. A slightly more general algorithm theory and corresponding proof for the general case in which the fixpoint function changes with each iteration can be found in [Pavlovic10]. The same algorithm scheme is used, but SBFixpointIterationWorksetTheory is generalized and the proof shows that the fixpoint algorithm converges to a nonleast fixpoint in general.

The process of applying an algorithm theory is as follows: the goal is to generate a refinement of a given specification S and we desire to apply algorithm theory fixpoint. We construct a morphism

m:SBFixpointIterationWorksetTheory → S.

m is called a *classification* morphism [SmithD9305] because it explicates how S has a problem that can be treated by fixpoint iteration. To obtain an algorithm design, we then take the pushout of fixpoint and m. The effect is to instantiate the schematic definitions in SBFIW_Algorithm to the details of the problem in S.

The next question is how to emit a proof as a by-product of the pushout. One approach that we prototyped involves locales in Isabelle. A locale is a parametric proof, which allows it to be instantiated. Since the algorithm theory SBFIW_Algorithm is parametric

on SBFixpointIterationWorksetTheory, a locale that captures the reasoning above can be instantiated when we use a pushout to instantiate SBFIW_Algorithm.

One challenge in formulating a precise abstract specification for workset-based fixpoint iteration is the inherent nondeterminacy of selecting what to do next from the workset – the order of selection doesn't matter as far as the final result is concerned because the iteration computes a function. However, the semantics of Specware's MetaSlang language is classical, in terms of sets and functions, so nondeterministic selection from a set is not a function and is therefore not allowed. This has been a longstanding problem in the formal specifications community.

We discovered a novel solution to this problem by exploiting the black-box nature of coalgebraic types (called cotypes or codatatypes in the literature). For our Garbage Collection (GC) examples, we require a workset variable which helps control the iterative process of tracing live nodes. An operation that works directly on the workset to extract an element cannot be a function. Instead, the trick is (1) to express the workset as an observation of the State (or Heap) cotype, and (2) to specify a select operation as a function of the State rather than the workset directly. That way, we can ultimately refine the select operation to a function that works on the set representation (e.g. a list) rather than the (abstract) set itself (see the section below on the Observer Refinement transformation). Solving this problem allows us much more freedom to develop specifications at the most abstract level possible, which allows simpler inference calculations and maximizes our implementation freedom.

Example: Finding Live Nodes

We apply the SBFIW_Algorithm algorithm theory to the problem of finding the live nodes in a heap.

```
liveasFIP = spec
  import Collector
  op FHeap(H:Heap)(ns:Set Node):Set Node =
     (roots H) \/ (allOutNodes H ns)
  end

live_as_fixpoint = morphism
SBFixpointIterationWorksetTheory -> liveasFIP
   {State      ⟼ Heap,
    pre        ⟼ findLive_pre,
    post       ⟼ findLive_post,
    p          ⟼ findLive,
```

```
X          ⟼ Node,
obs        ⟼ black,
nextState  ⟼ insertBlack,
F          ⟼ FHeap }
```

A classification morphism can be defined as in the figure. The first four translations of the morphism identify the SSP problem to solve. Conceptually, this part of the morphism can be obtained based on simple parsing of the the specification once it is given that findLive is the problem at hand. Since the goal of findLive is to find a set of Nodes, then X translates to Node. The abstract observer obs observes the growing set of nodes that are found to be reachable/live, so its image is the black observer and it is undated using insertBlack.

Since Specware's morphisms maps symbols to symbols, there is sometimes the need to construct a definitional extension to provide the requisite symbols for the codomain of a morphism. Here we extend the specification Collector with a defined function FHeap that serves as the image of the monotone function for the algorithm theory.

Taking the pushout of live_as_fixpoint and fixpoint SBFixpointIterationWorksetTheory -> SBFIW_Algorithm essentially yields an extension of Collector with a definition for findLive.

```
Collector1 = spec
  import Heap
  ...
  op WS (st:State): Set(Node) =  FHeap st (black st) -- black st
  op selectWS (st:State):
      {(st',ox): State * Option(X) |
               if WS st = empty_set
               then WS st' = WS st
                   &&  ox = None
               else ex(y:X)(y in? WS st
                        && WS st' = delete(y, WS st)
                        && ox = Some y)}

  op findLive(st:State | black st = supply st): {st':State | live st' subset black st' } =
      f_iterate st

  op f_iterate (st:State):State =
    case selectWS st
```

```
    of (st', None)   -> st'
     | (st', Some y) -> f_iterate(nextState st' y)

  theorem correctness_of_p is
    fa (st:State,st':State)
       (black st = supply st && st' = p st && live st' subset black st')
end-spec
```

Other examples include: finding primes via Sieve of Erastosthenes, reachability in a graph, dominators, constraint propagation, and many other problems.

Various extensions can be made to the fixpoint algorithm theory.

- Phase-based iteration -- We introduce a flag that signals when the state-based iteration is ongoing. This allows us to assert that an iteration-relevant invariant is to be maintained during iteration, and not at other times.

- Dynamic Fixpoint Iteration -- Note that in the formulation of SBFixpointIterationWorksetTheory, the monotone function F depends on the state. When the fixpoint algorithm is executing concurrently with an application that changes the state, then F itself may change. In that situation, what does it mean to reach a fixpoint? Under mild conditions, we showed in [Pavlovic10] that the generated fixpoint is a non-least fixpoint of the F based on the initial state. In a powerset lattice, the dynamic fixpoint generates a supserset of the least fixpoint. For garbage collection, this means that the collector generates a superset of live nodes, or conversely, a subset of dead nodes. This is sound, since it satisfies the safety condition of garbage collection: never collect live nodes.

3.6.2 Transformations for coalgebraic specifications

In the following subsections we describe transformations that are specific to coalgebraic specifications. These are new transformations that we developed for our GC derivations.

3.6.3 Observer and Transformer Introduction

During the derivation process, there are various cases in which we need to introduce new observers and transformers. Sometimes this comes about as part of the normal development process of a library theory. For example, the domain theory for garbage

collection starts with mutable directed graphs as a way to specify heaps, with Graph as the basic cotype, and the observers and transformers discussed earlier. Graphs are then extended to Heaps by adding an observer for roots, supply nodes, and ghost observers for live, dead, and active nodes. New transformers include addRoot (to add a new root; e.g. a reference from a register) and addSupply (to add a dead node to the supply list).

One well-formedness obligation in coalgebraic specifications is that each transformer must specify how it affects all observers. Consequently, when we extend a specification with a new observer, then care must be taken to incrementally add coinductive constraints to all transformers that affect it. We extended Specware to allow incremental accumulation of coinductive constraints:

```
refine def newGraph (ni: NewInfo)
  : {H: Heap | roots H = empty_set
         && supply H = initNodeIds ni }
```

which adds two constraints to the (previously introduced) specification for transformer newGraph. In words, the roots of a Heap are initialized to the empty set and the supply nodes are initially all nodes.

Another case that introduces new observers is in algorithm theories. For example, the state-based fixpoint iteration theory introduces the workset as an observer of state, together with an invariant definition. As discussed later, the invariant is eagerly maintained by the Observer Maintenance transformation so that for each transformer, the coinductive constraint for the observer is automatically calculated (and the calculation is emitted as a proof of correctness for the refinement).

The Observer Maintenance transformation (see next subsection) is another explicit mechanism for introducing observers. The arguments to the transformation include a new observer of state and an invariant that characterizes the observer in terms of the current value of other observers.

3.6.4 Observer Maintenance

Recall that the algorithm theory of the previous section, SBFIW_Algorithm, introduced an observer WS that is intended as the workset of the fixpoint iteration; i.e. the frontier of elements that are candidates to add to the growing fixpoint. The observer has a definition, which allows us to compute its value on demand, but there are situations where performance can be increased by incrementally maintaining the observer rather than recomputing it from scratch. The characteristic scenario is the occurrence of WS in a loop where its value is incremented once per iteration.

The Observer Maintenance transformation is applied to a defined observer, say

obsE: State → E,

that we desire to maintain incrementally rather than compute on demand. The performance improvement comes from a space-time tradeoff: we store the incrementally computed value of the observer obsE so that, on demand, we can simply access its value (knowing that the stored value equals the defined value).

In the context of a derivation, the idiom is that we introduce a fresh observer and its definition

```
op obsE(st:State):E = (phi st).
```

Rather than manually enter the coinductive constraints that assert how each transformer affects this observer, we wish to use the definition to automatically calculate those constraints and add them. In a Specware metaprogram/derivation-script, we write

```
transform S by {maintain(obsE), ... other transformation cmds}
```

The Observer Maintenance transformation performs the following steps:

1. for each undefined transformer

```
op t(st:State | pre st)(args:Args):{st':State | post st args st'}
```

augment its pre- and post-conditions with the obs invariant as follows:

```
op t(st:State | pre st && obsE st = e st)(args:Args):
    {st':State | post st args st' && obsE st' = e st'}
```

2. apply simplification rules in context to normalize it to the coinductive form

```
obsE st' = delta st' (obsE st) )
```

for some function delta:State->E->Ed}.

3. refine the specification of t to

```
op t(st:State | pre st && obsE st = e st)(args:Args):
    {st':State | post st args st' && obsE st' = delta st' (obsE st)}
```

Note: Step 1 applies only to transformers that are specified but do not yet have a definition. Once a transformer has a definition in terms of other transformers, then the definition body presumably maintains the observer invariant by construction.

In garbage collection derivations, the observer maintenance has a variety of uses. One is to incrementally maintain the workset in the fixpoint iteration that traces the live nodes.

3.6.4.1 Example: Maintaining the workset in a tracing algorithm

Continuing our example from earlier, notice that the algorithm theory has introduced a new observer, WS, with its definition. The fact that there are frequent calls to this observer during the iterative loop by selectWS suggests to apply the Observer Maintenance transformation.

```
OM1 = transform Collector1 by
      {maintain(WS)
        [... theorems-to-apply ...]}
```

We extend our derivation script/metaprogram to apply Observer Maintenance to observer WS in the Collector1 specification, as shown. The optional argument to maintain(WS) is a list of theorems that can be used in the calculations of the update codes. More generally, it is a specification of theorems to apply, focusing commands, and other calculation tools to apply, such as common-subexpression elimination and simplification.

The transformation first collects all undefined transformers (of the cotype that WS depends on), and then attempts to calculate incremental update code to maintain the observer's invariant. We show two examples by way of illustration: insertBlack and addNode.

To maintain the invariant

```
WS st = FHeap st (black st) -- black st
```

the specification for transformer insertBlack

```
op insertBlack(st :State)(n:NodeId | n in? nodes st):
  {st':State | black st' = insert(n, black st)}
```

is transformed to

```
op insertBlack(st :State | WS st = FHeap st (black st) -- black st)
                (n:NodeId | n in? nodes st):
                {st':State | black st' = insert(n, black st)
                        &&  WS st' = FHeap st' (black st') -- black st'}
```

We will need the theorem

```
theorem distribute_allOutNodes_over_insert is
  fa(G:Graph, n:NodeId, ns:Set NodeId)
    (allOutNodes G (insert(n, ns)) = allOutNodes G ns \/ (outNodes G n))
```

We then simplify the postcondition on WS st', seeking to derive an incremental, coinductive form for it. As assumptions, we gather (1) the preconditions, (2) other the contextual postconditions (on black st'), and (3) any implicit frame conditions (observers that do not change under the transformer).

```
theorem distribute_set_diff_over_union is [a]
  fa(A:Set a,B:Set a,C:Set a)
    ((A \/ B) -- C = (A -- C) \/ (B -- C))

theorem distribute_set_delete_union2 is [a]
  fa(A:Set a,B:Set a,y:a)
    (~(y in? B) => set_delete(y, A \/ B) = set_delete(y, A) \/ B)

theorem distribute_set_diff_over_right_insert is [a]
  fa(c:Set a,d:Set a,y:a) (c -- set_insert(y,d) = set_delete(y, c -- d))
```

```
Assume: WS st = FHeap st (black st) -- black st)
          &&  n in? nodes st
          &&  st' = insertBlack st n
           && black st' = insert(n, black st)
           && roots st' = roots st
```

Simplify: WS st'

```
  =   { by assumption on WS }
```

FHeap st' (black st') -- black st'

= { applying the assumption on black }

FHeap st' (insert(n, black st)) -- (insert(n, black st))

= { unfolding the definition of FHeap (from liveasFIP) }

(roots st') \/ (allOutNodes st' (insert(n, black st))) -- (insert(n, black st))

= { apply assumption about roots }

(roots st') \/ (allOutNodes st' (insert(n, black st))) -- (insert(n, black st))

= { apply assumption about st' }

(roots st) \/ (allOutNodes (insertBlack st n) (insert(n, black st))) -- (insert(n, black st))

= { apply theorem allOutNodes_of_insertBlack }

(roots st) \/ (allOutNodes st (insert(n, black st))) -- (insert(n, black st))

= { applying distribute_allOutNodes_over_insert }

((roots st) \/ (allOutNodes st (black st)) \/ (outNodes st n)) -- (insert(n, black st))

= { applying distribute_set_diff_over_union }

((roots st) \/ (allOutNodes st (black st)) -- (insert(n, black st)))
\/
((outNodes st n) -- (insert(n, black st)))

= { fold the WS invariant }

WS st \/ ((outNodes st n) -- (insert(n, black st)))

= { applying theorem distribute_set_diff_over_right_insert }

WS st \/ delete(n, (outNodes st n) -- (black st)).

As a result of this calculation, we can refine the specification for insertBlack to

```
op insertBlack(st :State | WS st = FHeap st (black st) -- black st)
                (n:NodeId | n in? nodes st):
                {st':State | black st' = insert(n, black st)
                        &&  WS st' = WS st \/ delete(n, (outNodes st n) -- (black st))'}
```

We treat this as an atomic action, since we require that no other process/thread can observe a state in which the invariant on WS is violated. Notice that this is the essential update underlying Dijkstra's on-the-fly concurrent mark&sweep algorithm, which was discovered after many flawed attempts [Dijkstra78]. The essence falls out by a simple calculation in our setting.

Note that insertBlack is a Collector operation. We also perform Observer Maintenance on Mutator transformers. Effectively, this requires that the Mutator cooperate with the Collector in maintaining the workset. We show this in the following calculations for addNode or setTgt.

3.6.4.2 Example: Reference Counting

As another of the many examples of applying Obsever Maintenance, consider the maintenance of a reference count observer.

```
op  refcnt(G:Graph)(n:Node):Nat = occs(n,roots G) + inArcCnt G n (arcs G)
```

and its maintenance with respect to a simple addArc transformer:

```
op  addArc(G:Graph| refcnt G n = occs(n,roots G) + inArcCnt G n (arcs G))
            (x:Node, y:Node) :
            {G':Graph | nodes G' = nodes G
                    ∧ outArcs G' x  = (outArcs G x) + (x→y)
                    ∧ refcnt G' n = occs(n,roots G') + inArcCnt G' n (arcs G')}
```

The simplification of the inserted occurrence of refcnt in the postcondition is

```
refcnt G' n = occs(n,roots G') + inArcCnt G' n (arcs G')
            = occs(n,roots G∪{x→y} + inArcCnt G∪{x→y} n (arcs G∪{x→y})
            = if n = y
                then 1 + occs(n,roots G) + inArcCnt G y (arcs G)
```

```
        else occs(n,roots G) + inArcCnt G y (arcs G)
= if n = y
     then 1 + (refcnt G n)
    else (refcnt G n).
```

3.6.4.3 Other Examples

We used Observer Maintenance extensively in our GC derivations. It gives rise to natural data structures, their meaning, and efficient incremental computation. In particular, the following data structures emerge from Observer Maintenance:

1. Reference Count: reference count, supply
2. Mark&Sweep: workset, root count, supply length, supply
3. Copying Collector: new-space, root count, supply
4. Generational Collector: new-generation, root count, supply

The following two subsections describe generalizations of Observer Maintenance.

3.6.4.4 Generalization of Observer Maintenance: Conditional Invariants

We developed a technique for maintaining conditional invariants. The motivation is that for efficiency's sake we want to maintain an invariant during the marking phase, but not during other phases (e.g. sweeping and when the collector is idle). To our knowledge, all previous work on transformations to maintain invariants has focused on global state invariants (which are required to hold in all states at all times). The invariant that characterizes the workset in the tracing of live nodes only needs to hold during the Marking phase. Any work done to maintain the workset during sweeping, or when the Collector is idle, is wasted. Our approach is to have a globally observable flag *marking?* that is on exactly during the Marking phase. Then the invariant can be expressed as a conditional:

$$marking? \Rightarrow invariant.$$

Our transformation then calculates how to incrementally maintain the invariant over a transformer. For each transformer, we specify whether the flag can be assumed on (or off) for its duration, if knowable at specification-time.

3.6.4.5 Generalization of Observer Maintenance: Maintaining an Inequality

Consider again the workset invariant:

WS st = FHeap st (black st) -- black st)
= (roots st) ⋁ (allOutNodes H (black st)) -- black st

The action of a concurrent application will change the heap, with the effect of changing the set of root nodes and changing the outnodes of various live nodes. This will typically cause the invariant to be violated.

Various extensions can be made to the Observer Maintenance transformation. In particular, it is sometimes necessary to maintain an inequality, rather than an equational invariant. There are two main reasons for weakening equational invariants to inequation[3]. The intuition is that the action of a concurrent process/thread can cause the weakening of an inequation, instead of the outright breaking of an equation.

First, we may not have strong enough lemmas to calculate code to maintain the equality, but enough to maintain an inequality. Second, we may have strong enough lemmas to calculate maintenance code for an equation, but it is too expensive to compute. It may be less expensive to maintain an inequality and then use a residual check at runtime to eliminate the over-approximative delta (as in Workset case). In concurrent algorithms, the distinction of the cases is moot. The actions of a concurrent agent often mean that we can only know and enforce at design-time an inequation. Related to both of these points, it is often noted that inequalities arise frequently in concurrent algorithm design. It is common that where an equation is enforceable in a sequential algorithm, its concurrent variant requires an inequation due to interference between threads/processes.

A good example of this is the workset in a Concurrent GC. In the sequential/stop-the-world collector, we can maintain the exact workset given by WS above. In a concurrent setting, where the Mutator may swing pointers during the process of finding live nodes, it would be prohibitively expensive to maintain the exact frontier of reachable live nodes. The only efficient solution is to maintain

invariant (FHeap st (black st) -- black st) <= WS st

i.e. that we must maintain an upper bound on the frontier of reachable nodes that have not yet been marked as live.

For example, consider worksets in dynamic fixpoint algorithms. The best we can do in a dynamic fixpoint is find an overapproximation of the least fixpoint. Consequently, the

[3] In Rely-Guarantee this is called stabilization — weakening the invariant so that it is stable under state changes by the environment.

workset, although ideally defined by an equality (set difference), can be weakened to an inequality; i.e. as long as way can incrementally compute an upper bound on the ideal WS, then we preserve the property of converging to a (nonleast) fixpoint. This happens here in the maintenance of WS under the swinging of an arc/ptr - we may leave a dead node in the WS, resulting in an overapproximation of the currently live nodes. The prescription for a bounding WS (vs exact equality) should fall out of the overall spec and the algorithm theory (which introduces the workset).

FD in the dynamic fixpoint setting - conjecture: in a static fixpoint setting, we can use an upper bound of the workset to converge to a nonleast fixpoint. In a dynamic setting, this may be necessary for performance reasons (e.g. selectWS which we generalize from a set to a multiset).

In the Specware collector derivations, we noted the need for both kinds of inequality maintenance. We found a way to produce the correct effect in the derivations, but not in the most general way. The correct approach is to generalize the Observer Maintenance transformation to allow calculation of update code for invariants of the form

```
invariant  E st <= obs st
```

where <= is a partial order, and the invariant provides a (upper or lower) bound on the observer obs. Equational invariants are a special case. The goal of the Observer Maintenance transformation is to calculate update code that maintains as strong a bound on the observer as possible, given what is knowable statically (at design-time).

3.6.4.6 Related Work

The observer maintenance transformation builds on earlier work on strength reduction in compilers, finite differencing [Paige82], incrementalization [Liu13]. These previous transformations work by looking up the update code from pre-computed tables. Consistent with our generalization of Paige's Finite Differencing transformation [SmithD9009], we allow the maintenance of invariants over user-defined vocabulary, since we calculate the update code in the context of the application domain theory; that is, we use the axioms and theorems of the domain as part of the calculation of update code. Observer maintenance can be viewed as an adaptation of our generalization of finite differencing to coalgebraic specifications.

The well-known "tricolor" abstraction invented by Dijkstra et al. [Dijkstra78] arises naturally as a by-product of a generic (i.e. problem-independent) transformation (and calculations) for maintaining invariants. The "gray" nodes correspond to the workset

that is used to control the fixpoint iteration process. The "black" nodes are the marked nodes (nodes that are, or were, live), and the white nodes are unmarked nodes. The necessity for mutator cooperation when assigning a reference (via addArc) falls out via calculation. The need for a write barrier falls out from the need to perform the FD increments atomically with their triggering action. Dijsktra's decision to leave newly allocated nodes white unless their parent is black also falls out by straightforward calculation.

One point is that there is no need for intricate problem-specific conceptualization and ad-hoc reasoning during design - the design concepts and inferences are generic in their outline and are only problem-specific in that they rely on problem-specific requirements/goals and problem-specific axioms and theorems. That is, the designs are generic but tailored by generic inference patterns to the specified problem.

3.6.5 Observer Refinement

A key problem in formal specifications has been how to refine datatypes from their initial abstract form to their final concrete form. Typically the abstract form allows simpler reasoning during the design process, and the concrete form is complex but provides good performance. Traditionally, refinement processes (in both the imperative and functional language communities) have focused on refining (or reifying) abstract types to concrete types. There are well-known techniques for doing that when the abstract types are defined. However, in the coalgebraic style that we are developing the abstract cotypes have no definition until the last step of refinement before code generation. Our breakthrough was realizing that cotypes are characterized by observers and it is the observers that must be refined. One way to think about this is that rather than refining the abstract cotype directly, we refine various observations of it, thereby indirectly refining the cotype. At the last step of refinement, we define the cotype as a product (record, struct) of the concrete observers via the finalizeCotype transformation (see later section).

We reduced these insights to practice by defining a simple transformation on coalgebraic specifications, called ***observer refinement***, which we now summarize.

The context is that we have an existing observer obsE:State→E and we have its effect on various transformers stated coinductively in their postconditions. We wish to refine obsE to an observer of a more concrete type, and we do so by introducing a new observer

```
op obsEC:State→EC
```

of a more concrete type EC, together with an invariant that shows how to abstract EC observations to E observations:

```
axiom obsE_invariant is
    obsE st = abs obsEC st
```

where abs:EC→E is an abstraction function. To make the calculations work it is useful, maybe necessary, for abs to be a homomorphism from EC to E.

The goal is to eliminate obsE in favor the more concrete observation obsEC. We use the invariant obsE_invariant to replace obsE everywhere in the postconditions of transformers and observers, and then simplify.

We implemented the following syntax for observer refinement in a Specware tactic/metaprogram:

```
transform S by {refine(obsE, obsE_invariant),  ... simplification rules to use …}
```

The refine transform calculates the following:

1. for each transformer t:State→Args→State that is coinductively specified by the form

```
op t(st:State)(a:Args|pre(st,a)):
    {st':State |   ...  & obsE st' = upsilon st a (obsE st)}
```

2. unfold the def of obsE (i.e. apply the invariant) yielding

```
op t(st:State)(a:Args|pre(st,a)):
    {st':State |  ...  & abs obsEC st' = upsilon st a (abs obsEC st)}
```

3. then calculate a sufficient condition (since we can strengthen a postcondition in a refinement) to get a refined definition of the form

```
op t(st:State)(a:Args|pre(st,a)):
      {st':State |  ... && obsEC st' = chi st a (obsEC st)}
```

for some function chi: State → Args → EC → EC.

More generally, we can replace all occurrences of obsE by obsEC in axioms, theorems, pre/post-conditions, and definition bodies; and then simplify. Later, at code-generation time, there will be no references to obsE, so it is effectively eliminated in favor of obsEC.

This transformation is used to reformulate the specification in terms of more concrete observers. One unexpected fallout of this technique is that we can specify an observer that extracts an element of a set, which is not possible in a purely algebraic setting.

3.6.5.1 Example: Refining the WorkSet to a WorkList

In the previous section we applied Observer Maintenance to the Workset observer WS, which has type

```
op WS: Graph → Set NodeId.
```

For the sake of efficiency, we wish to implement WS by a List representation. We introduce a new observer

```
op WL: Graph → List NodeId.
```

and define

```
axiom WS_as_List is
  fa(G:Graph) WS G = List2Set (WL G)
```

where List2Set is a homomorphism from Lists to Sets; that is, we have laws such as

```
theorem List2Set _Nil is [a]
    (List2Set (Nil) = (empty_set:Set a))

theorem List2Set _Cons is [a]
   fa(y:a,lst:List a) (List2Set (Cons(y,lst)) = set_insert(y, List2Set lst) )

theorem List2Set_comprehension is [a]
   fa(p:a->Boolean) (x:a | p}) = List2Set( [ x:a | p ])

theorem List2Set_concat is [a]
   fa(l1:List a, l2:List a) List2Set(l1++l2) = List2Set(l1) ∪ List2Set(l1)
```

We apply Observer Refinement by including a statement

```
transform Collector2 by
    { implement(WS,WS_as_List)
        [rl _. List2Set _Nil,
         rl _. List2Set _Cons,
         ...]}
```

The effect is to

1. replace all occurrences of WS by List2Set∘WL
2. simplify all such occurrences, replacing references-to and updates-of WS by WL

For example, returning to our simple addArc transformer which maintains the WS invariant

```
op  addArc(G:Graph | WS G= (roots G ∪ outArcs G (black G))\(black G))
          (x:Node, y:Node) :
          {G':Graph |        nodes G' = nodes G
                        ∧ outArcs G' x  = (outArcs G x) + (x→y)
                        ∧ WS G' = WS G ∪ {y | x∈black G ∧ y∉black G}}
```

the transformation automatically tries to simplify the update to WS as follows:

Assume: fa(G:Graph) WS G = List2Set (WL G)

Simplify: WS G' = WS G ∪ {y | x∈black G ∧ y∉black G}

= { by assumption on WS }

List2Set(WL G') = List2Set(WL G) ∪ {y | x∈black G ∧ y∉black G}

= { List2Set_comprehension }

List2Set(WL G') = List2Set(WL G) ∪ List2Set ([y | x∈black G ∧ y∉black G])

= { List2Set_concat }

List2Set(WL G') = List2Set(WL G ++ [y | x∈black G ∧ y∉black G])

⟸ { Leibniz/substitutivity }

WL G' = WL G ++ [y | x∈black G ∧ y∉black G].

We can then refine the specification to

```
op  addArc(G:Graph) (x:Node, y:Node) :
        {G':Graph |      nodes G' = nodes G
                  ∧ outArcs G' x  = (outArcs G x) + (x→y)
                        ∧ WL G' = WL G ++ [y | x∈black & y ∉ black]  }
```

which allows us to maintain the workset as a list. In our derivations we go on to maintain the list as a stack.

The Observer Refinement transformation is used extensively in our GC derivations and has been critical to obtaining good performance. It allows us to develop domain specifications and problem formulations in terms of abstract types such as sets and functions, knowing that we can systematically and correctly refine them to efficient implementation types.

3.6.5.2 Example: Extracting an Arbitrary Element of a Set

Another example illustrates how Observer Refinement solves a long-standing problem in formal refinement, namely, how to extract an arbitrary element of a set. In the fixpoint algorithm, a central step is selecting an element from the Workset WS and finding its outArcs. In a purely functional world the only way to select an element from a set is essentially to impose a linear order and select the minimum element. Intuitively, this is should an easy and natural operation, at least on finite structures, and it provides a good example of the mathematical formalism hindering rather than helping us. Our approach uses the coalgebraic setting – the Workset WS is an observation of the State/Graph, which has unknown structure. Under Observer Refinement, we indirectly refines the structure of State, ultimately allowing us to define a functions that pulls an element out of the workset in constant time.

The arb observer is specified to return an element from a nonempty workset

```
op  arb(G:Graph | WS G ≠ {}): {n:NodeId | n ∈ WS G }.
```

Using the following theorems from the library

theorem List2Set_element is [a]
fa(lst:List a,n:a) n ∈ List2Set(lst) = (n ∈ lst)

theorem List_element is [a]
fa(lst:List a) lst ≠ {} ⇒ first(lst) ∈ lst

we apply Observer Refinement to refine WS to WL, we get the following calculation:

Assume: WS G = List2Set (WL G),
WS G ≠ {}
Simplify: n ∈ WS G

= { by assumption on WS }

n ∈ List2Set(WL G)

= { by List2Set_element }

n ∈ WL G

⇐ { List_element }

WL G ≠ [] ⇒ n = first(WL G)

= { discharging the antecedent by assumptions }

n = first(WL G).

So the specification for observer arb is refined to

op arb(G:Graph | WS G ≠ {}): {n:NodeId | n = first(WL G) }.

One way to think of why this works is to consider a cotype State as an initially undefined implementation state, which has various observers. As we perform observer refinement steps, we get closer to a definition of State that allows efficient observations and state transformations. This allows us to start a derivation with very abstraction observers with unknown implementation, then to incrementally add implementation constraints.

3.6.5.3 Related Work

Until recently, the only mechanism for refining abstract datatypes in Specware was to apply library specification morphisms for various datatype refinements; e.g. Sets_to_Lists, and Maps_to_Lists [Blaine94]. These morphisms entail the need for quotient types (e.g. Sets are refined to a quotient type over Lists) and predicate subtypes (e.g. Sets are refined to a predicate subtype of Bags). Observer Refinement adds another method for refining abstract types that provides more flexibility and leads to better performance. One difference is that OR only allows the refinement of the type of an observer, rather than every occurrence of an abstract type.

Observer Refinement is related to datatype refinement as first defined by Hoare in 1972 and subsequently generalized [He86]. In the (similar) data reification of VDM, it is required that the homomorphism (called the retrieve function) must be surjective. In other words, each abstract value has at least one concrete representation.

3.6.6 StructureEx

This transformation eliminates quantifiers in favor of let-bindings and substitutions. It plays a crucial role in translating logical postconditions into a more functional form. We developed this transformation and made many extension to handle cases.

3.6.7 FinalizeCotype: Cotype Definition and Postcondition Synthesis

During a derivation, we typically introduce a cotype without a definition, but add observers to it in subsequent refinement steps.

Observers at any stage in the refinement process come in several flavors. Some observers have a definition (and so they are eagerly computed when needed). Some are undefined but are specified by their effect on various transformers. Some observers have an invariant characterization and are incrementally computed via the Observer Maintenance transformation. Some are ghost observers and therefore have no effect on computation, since they exist solely to increase the precision of system properties.

The finalizeCotype transformation is a packaging of two related transformations: cotype definition and postconditions synthesis.

3.6.7.1 Transformation: Cotype Definition

The cotype definition transformation introduces a definition for the cotype as a tuple, or record named fields. It works by collecting the undefined observers that are not ghosts

and making them the fields of the tuple. It then gives a definition to each observer as a field access to the local cotype element (commonly the state).

Refinements of a coalgebraic specification correspond to subclassing. If we refine a spec Sspec introducing cotype S to a spec TSspec that introduces additional observers and transformers on S, then any S operator can be applied to any T object. This is useful for example in refining the Graph/Heap notion of Node to "subclasses" Register, StackNode, HeapNode, and Supply.

Suppose that in a refined spec later in the derivation of a GC, we have these undefined or maintained observers

```
nodesL    : Graph -> List Node
rootsL    : Graph -> List Node
supply    : Graph -> List Node
WL        : Graph -> List Node
blackCM   : Graph -> Map(Node,Boolean)
tgtIM     : Graph -> Map(Node,Map(Index, Node))
```

We implemented the following syntax for defining a cotype in a Specware tactic/metaprogram:

```
transform S by { finalizeCoType(Graph)}
```

The transform analyzes the spec S and produces a refined specification with the following definition:

```
type Graph= { nodesL   : List Node,
              rootsL   : List Node,
              supplyL  : List Node,
              WL       : List Node,
              blackCM  : Map(Node, Boolean),
              tgtIM    : Map(Node, Map(Index, Node))
              rootCount : Nat
              }
```

The cotype definition transformation also gives definitions to the observers that are packaged up in the record (e.g. nodesL, rootsL, etc.),

```
op nodesL(G:Graph): List Node = G.nodesL
op rootsL(G:Graph): List Node = G.rootsL
```

and so on. The transformation also unfolds calls to them everywhere, eliminating them as functions. For example

```
op addRoot (G: Graph)(n: NodeId | n in? G.nodesL)
        : {G': Graph | G'.rootsL = n :: G.rootsL
                && G'.WStack = if Map.TMApply(G.blackCM, n)
                                then G.WStack
                                else push(n, G.WStack)
                && G'.rootCount = 1 + G.rootCount }
```

3.6.7.2 Synthesize transformers from postconditions

The second part of the finalizeCotype transformation, synthesizes definitions for each transformer. It does so by translating the coinductive constraints in the postconditions of transformers into update of the newly-introduced cotype record. Continuing the example,

```
op addRoot (G: Graph)(n: NodeId | n in? G.nodesL)
        : {G': Graph | G'.rootsL = n :: G.rootsL
                && G'.WStack = if Map.TMApply(G.blackCM, n)
                                then G.WStack
                                else push(n, G.WStack)
                && G'.rootCount = 1 + G.rootCount } =
    G << { rootsL = n :: G.rootsL,
           WStack = if Map.TMApply(G.blackCM, n)
                    then G.WStack
                    else push(n, G.WStack),
           rootCount = 1 + G.rootCount}
```

where G << {f1 = a, f2 = b, …, fn = z} is a Metaslang operation that denotes a record G' in which each field of G' is the same as in G, except G'.f1=a, G'.f2=b, …, G.fn=z. The right-hand sides of the equations are evaluated first, and then the changes are made. Note that this transformation introduces a functional definition of the state change – it computes the new state as a function of the old/input state.

3.6.8 Globalization

We implemented a new transformation that performs *Globalization*. Its effect is to transform the implicit state in a coalgebraic specification to explicit global/shared state. It allows us to generate truly imperative code.

Globalization can be described via the following abstract example. Here the cotype State has been defined as a pair and the observer c and transformers f and g are single-threaded on State; i.e. they take State as input and produce a State as output.

```
S = spec
 type State = {a:A, b:B}
 op c(st:State):C = h(st.a, st.b)
 op f(st:State)(arg:Arg): State =
    st << {a = alpha st.a, b = beta st.b}
 op g(st:State)(arg:Arg): State*D =
    (st << {a = gamma st.a, b = delta st.b},
     eps (st.a) (st.b))
end-spec
```

We implemented the following syntax for globalizing a cotype in a Specware tactic/metaprogram:

```
transform S by {globalize(State)}
```

The Globalization transformation on a cotype State requires that the type be single-threaded; i.e. such that there can be no two elements of the type simultaneously live during execution. Single-threadedness can be detected statically, however, the finalizeCotype transformation produces single-threaded definitions, and so it provides suitable input to the Globalization transformation.

Since Specware's Metaslang language is functional, and has no notion of state, the Globalization transformation necessarily is a step from Metaslang toward an imperative language, CommonLisp and C in our case. Its steps are to

1. Introduce a global variable of the cotype, say, var st:State.
2. For each observer and transform, eliminate State as an explicit parameter and return, and replace local references to state by global references.
3. Replace record updates of the cotype by assignments

Shown in a pseudo-imperative notation, the effect of Globalization on S is

```
type State = {a:A, b:B}
var st:State

op c():C = h(st.a, st.b)
op f(arg:Arg): Unit =
   (  st.a := alpha st.a
   || st.b := beta st.b )

op g(arg:Arg): D =
   (  st.a := gamma st.a
   || st.b := delta st.b
   || return (eps (st.a) (st.b))
   )
```

The effect of Globalization is to introduce a global variable st of cotype State, and all accesses to st are now to the global (versus access to the parameter as before the transformation) and changes to fields of State are via destructive assignment rather than functional copy&modify. Our ad-hoc notation here treats concurrent assignment statements in an atomic region, in order that invariants are not observed to be violated.

One technical issue arose during implementation. The key assumption of globalization is that a State parameter occurs single-threaded throughout the specification. As it turns out, the single-threadedness property holds for the Collector and Mutator (application-oriented) parts of the specification, but the part of the specification that gives the theory of States is not single-threaded – States are partially ordered to support a fixpoint iteration over them. Another counter-example is any theorem about the evolution of State; e.g. that the dead nodes are monotonically increasing in time, since this involves a relation between two consecutive states. The solution to this problem is to slice the specification according to the application-oriented definitions before applying globalization, keeping only parts of the code that are intended for execution as opposed to specifying properties of State and State evolution. This entailed the need to enhance our existing slicing mechanisms to cope with some of the new coalgebraic features of MetaSlang.

The correctness of this transformation is straightforward in a sequential imperative language. It is more difficult to prove in a concurrent setting because of possible interference with State as shared memory. The proof that this transformation works for concurrent applications depends on assume-guarantee reasoning. As discussed earlier, for concurrent garbage collection, we start with the specified assumption that the

Mutator cannot access the “black/white/gray” information about nodes, which is strictly Collector data – so the Collector can assume that other processes can only increase the set of white/dead nodes. Conversely, the Collector cannot modify the set of live/black nodes – so the Mutator can assume that other processes leave the set of live nodes is invariant. This general reasoning is what’s needed for the proof that Globalization applied to concurrent GC is correct.

One improvement to the Globalization Transformation was to generate updates to the global state that are maximally localized. In CommonLisp parlance, we replaced setq’s by setf’s. Extra analysis machinery and tables of setters and getters (updates and accessors) were needed.

A guiding concern in the above explorations has been whether the transformations can be simple and clean enough to emit proofs as a by-product; i.e. either we prove the transformation correct once-and-for-all or we generate a proof script justifying each application. Towards this end, we factored our Globalization transformation into a sequence of simpler transformations so that each could be extended to emit proofs with each application. The steps include (1) Linearize the single-threaded state updates (to get the code in a form that will avoid interference upon translation to sequential imperative form), (2) record merging (to normalize some expressions), (3) future usage count analysis (to determine when a value will be no longer used), and (4) translation to C99 abstract syntax with destructive updates as permitted by analysis. As of the end of project we had not completed proof-emission enhancements for these subtransformations.

3.7 Other Transformations

The transformations discussed in previous section were new in Crash and focused on transforming coalgebraic aspects of our specifications. Several other transformations were applied and further developed in the project, and we discuss those below.

3.7.1 Simplification

The most basic optimizing transformation is context-sensitive simplification. The idea is simple: an expression is simplified by first gathering contextual properties and then applying conditional equational rules to find a “simpler” form modulo context.

An interesting phenomenon was revealed during the project regarding the difference between simplification of algebraic/functional expressions and coalgebraic/state-

changing expressions. In functional expressions, the simplest form is another expression that takes less time/space to evaluate (at runtime) to a value. This kind of simplification is often carried out by symbolic evaluation at design-time, ideally to a constant. In state-changing expressions, the ideal is minimal state change. This distinction arose in our derivations and can be illustrated in the following example:

```
op f(st:State | obs st = 0): {st':State | obs st' = obs st}
```

where obs is a Nat valued observer of State. Our simplifier at first replaced the postcondition by

```
obs st' = 0
```

which ultimately is implemented as an assignment of 0 to the observer. However, this "simplification" ignores the information in the constraint that the observer is unchanged by f, so the postcondition as it stands is in simplest form. It should ultimately be implemented as a no-op in this case.

In the functional world, values are created from other values. In the side-effecting world, progress is characterized by making minimal changes to the current state.

3.7.2 Type Isomorphism

The type isomorphism transformation refines one type T into an isomorphic type T' with appropriate translations of operations involving T. We extended it to handle previously unhandled cases; specifically isomorphisms on types with multiple parameters, such as Map(a, b) and relaxing the requirement that the types be named types. Also in order to complete the isomorphism transformation in our garbage collection derivations we needed to supply extra distributive laws.

3.7.3 Partial Evaluation

We also implemented a *Partial Evaluation* transformation. Its effect is as follows. Suppose that we have a specification that includes a definition

```
def f(x:D):R =  G[e(0,x), x]
```

where f is defined by some term G that include a function call to e with a constant argument (here the constant is 0, but all that matters is that it is a constant). Partial Evaluation optimizes the definition by evaluating as much of it as possible at design time – by applying domain theorems to simplify terms. As a very simple example, if e were simple addition, then we could partially evaluate the expression 0 + x to x, resulting in (slightly) faster code.

3.8 Proof Emitting Transformations

Proof-emitting transformations was a key innovation that we began developing in the Crash project. Figure 6 illustrates/diagrams our approach to proof-emitting transformations and their role in generating refinements and proofs. The figure depicts the action of applying transformation T to specification A. The result is a generated refinement from A to B, represented by morphism σ. Moreover, the transformation generates a proof term that can be used to discharge the proof obligation of the refinement.

Conceptually, we treat a transformation as a mapping from (the abstract syntax of) a specification A to a triple that includes (1) a specification morphism σ, (2) the refined/target specification B, and (3) a proof term. The proof term is a summary of the calculations performed in generating B from A. Specware provides a general proof-obligation-generator utility that maps a specification morphism, such as σ, to a Metaslang formula that expresses its proof obligations (i.e. that the axioms of B imply the axioms of A modulo the translation induced by morphism σ). The intent of the proof term generated by a transformation is that it can discharge the proof obligations of the generated morphism.

The goal here is to have an independent proof-checker verify that the proof term generated by the transformation does indeed prove the obligations generated by the proof-obligation-generator. One feature of the structure of Figure 6 is that the left-hand side is independent of the proof-checker. We wanted the freedom to build translators to any proof-checker that was rich enough to express the Metaslang logic. As an independent proof-checker we chose Isabelle since we already had a partial translator from the Metaslang logic of Specware to the Isabelle/HOL logic. The following subsections describe several of the issues that arose in realizing this overall approach to generating proof-carrying code.

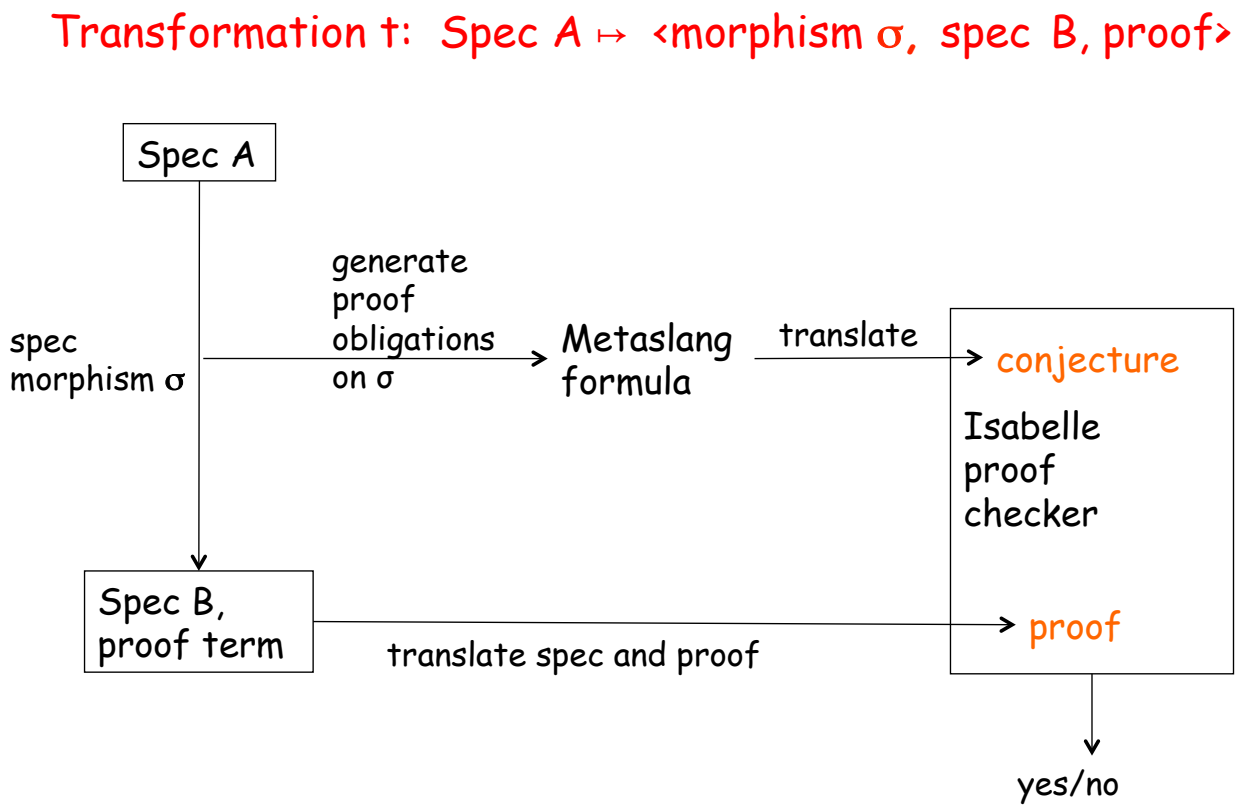


Figure 6: Proof-emitting Transformation

3.8.1 Instrumenting transformations to record calculation chains

We extended many of our library transformations to generate proof terms. During the course of the project we tried a sequence of approaches to the structure of the proof terms. Our first attempt was to record the sequence of equations used in a rewrite rule-based simplification. This was sufficient for several transformations, but couldn't handle the proofs involving recursive transformation of terms. Our second approach was to define transformation-specific datatypes to record transformation steps. After instrumenting several transformations this way, it became clear that there were many commonalities and we felt the need (and possibility to define) a uniform representation of calculations performed by transformations. Our third approach was to develop a uniform proof representation for all transformations. A portion of the definition of our proof term specification is:

```
type ProofInternal =
   | Proof_UnfoldDef (MSType * QualifiedId * MSVars * MSTerm * MSTerm)
   | Proof_EqSubterm (MSTerm * MSTerm * MSType * Path * ProofInternal)
   | Proof_EqSym ProofInternal
```

```
| Proof_EqTrans (MSType * MSTerm * List (ProofInternal * MSTerm))
| Proof_ImplTrans (MSTerm * ProofInternal * MSTerm * ProofInternal * MSTerm)
| Proof_ImplEq ProofInternal
| Proof_Cut (MSTerm * MSTerm * ProofInternal * ProofInternal)
| Proof_ImplIntro (MSTerm * MSTerm * String * ProofInternal)
| Proof_Assump (String * MSTerm)
| Proof_ForallE (Id * MSType * MSTerm * MSTerm* ProofInternal * ProofInternal)
| Proof_EqTrue (MSTerm * ProofInternal)
| Proof_Theorem (QualifiedId * MSTerm)
| Proof_Tactic (Tactic * MSTerm)
```

where, for example,

- Proof_UnfoldDef (T, qid, vars, M, N) is a proof that fa(vars) M=N at type T by unfolding the definition of qid,
- Proof_EqSubterm(M,N,T,p,pf) is a proof that M = N : T from a proof pf : M.p = N.p, where M.p is the subterm of M at path p
- Proof_EqSym(pf) is a proof that N=M from pf : M=N

and so on.

To give a sense of the details, consider the following rewrite steps performed in one of the Crash derivations:

```
{ 1: allOutNodes_of_addSupply }
     allOutNodes (addSupply H nid) (Set.set_insert(nid, black H))
—-> allOutNodes H (Set.set_insert(nid, black H))
{ 2: distribute_allOutNodes_over_set_insert }
      allOutNodes H (Set.set_insert(nid, black H))
—->  allOutNodes H (black H) \/ outNodes H nid
{ 3: Set.associative_union }
       roots H \/ (allOutNodes H (black H) \/ outNodes H nid)
—->  (roots H \/ allOutNodes H (black H)) \/ outNodes H nid
```

which is stored as the following proof term

```
EqTrans(Bool,
     roots H \/ allOutNodes H (black H) \/ outNodes H nid,
        [Sym(Theorem(Set.associative_union,
                roots H \/ (allOutNodes H (black H) \/ outNodes H nid)
                  = roots H \/ allOutNodes H (black H)  \/ outNodes H nid)),
              roots H \/ (allOutNodes H (black H) \/ outNodes H nid),
       EqSubterm(roots H \/ (allOutNodes H (black H) \/ outNodes H nid),
              roots H \/ allOutNodes H  (Set.set_insert(nid, black H)), Bool, [1],
```

```
                    Sym(Theorem(distribute_allOutNodes_over_set_insert,
                            allOutNodes H  (Set.set_insert(nid, black H))
                             = allOutNodes H (black H) \/ outNodes H nid))),
              roots H \/ allOutNodes H (Set.set_insert(nid, black H)),
         EqSubterm(roots H  \/ allOutNodes H
                       (Set.set_insert(nid, black H)),
                  roots H   \/ allOutNodes(addSupply H nid)
                       (Set.set_insert(nid, black H)), Bool, [1],
                  Sym(Theorem(allOutNodes_of_addSupply,
                           allOutNodes(addSupply H nid) (Set.set_insert(nid, black H))
                            = allOutNodes H  (Set.set_insert(nid, black H)))))),
                  roots H \/ allOutNodes(addSupply H nid)  (Set.set_insert(nid, black H))])
```

and is then rendered into the Isabelle/Isar proof-script language as Isabelle/Isar proof script

```
have subeq100:
    "(roots H \/ allOutNodes H (black H)) \/ outNodes H nid

      = roots H  \/ allOutNodes (addSupply H nid) (Set__set_insert(nid, black H))"
 proof -
  have "(roots H \/ allOutNodes H (black H)) \/ outNodes H nid
        = roots H  \/ (allOutNodes H (black H)  \/ outNodes H nid)"
   proof -
    have symeq95:
         "roots H \/ (allOutNodes H (black H)   \/ outNodes H nid)
       = (roots H \/ allOutNodes H (black H)) \/ outNodes H nid"
      proof -
       show "?thesis" by (auto simp only: Set__associative_union)
      qed
    show "?thesis" by (rule symeq95[symmetric])
   qed
  also
  have "... =  roots H  \/ allOutNodes H (Set__set_insert(nid, black H))"
   proof -
    have subeq97:
         "allOutNodes H (black H) \/ outNodes H nid
           = allOutNodes H (Set__set_insert(nid, black H))"
      proof -
       have symeq96:
            "allOutNodes H (Set__set_insert(nid, black H))
              = allOutNodes H (black H)   \/ outNodes H nid”
         proof -
          show "?thesis"
```

```
            by (auto simp only: distribute_allOutNodes_over_set_insert)
          qed
         show "?thesis" by (rule symeq96[symmetric])
        qed
       show "?thesis" by (rule arg_cong[OF subeq97])
     qed
…
  finally (HOL.trans)
 show "(roots H \/ allOutNodes H (black H)) \/ outNodes H nid
        = roots H \/ allOutNodes (addSupply H nid) (Set__set_insert(nid, black H))" .
qed
```

3.8.2 Translator from Metaslang logic to Isabelle logic

We extended an existing partial translator from Metaslang to Isabelle for two purposes. One was to translate refinement obligations and the other was to translate our proof terms into proof scripts that could be checked against the translated proof obligations.

Many aspects of the translation between these two higher-order logics were straightforward. However, completing this translator turned out to be trickier and take longer than expected. One key issue was translating Metaslang specs into Isabelle specs, and a special case is the translation of Metaslang formulas to Isabelle formulas. This was a source of ongoing difficulties since the Metaslang and Isabelle logics are similar but have many detailed differences. We worked on resolving two such differences: since Isabelle does not support predicate subtypes (including dependent types), we need to include the predicates from such types into the translation of a Metaslang expression, typically as an antecedent. We explored several variants of whether the antecedent should be normalized to the top level, or kept locally to preserve structure.

Another difference is that Isabelle does not support a name translation operation, while it is a basic operation on Metaslang specifications. This is a difficult feature to handle since the name translation must be applied recursively through the entire import structure of a specification. We completed work on handling the name translation operation on the Metaslang side. Since Isabelle doesn't have this feature, our translator from Metaslang to Isabelle had to perform a recursive copy-and-modify on the entire import structure of a specification and pass the whole structure to Isabelle, rather than appealing to Isabelle built-in specifications. This work was a part of the larger goal of supporting the generation of proofs that discharge automatically generated proof obligations for refinement steps. We continued to work on extending the Observer

Maintenance and Observer Implementation transformations to emit proofs at application time.

Another difference: We worked on improving the generation of Isabelle proofs from transformation sequences. The proofs include references to particular subterms that get transformed. These are indicated by their path from the root of the term. However, translation to Isabelle does not always preserve the term structure. In particular, in quantified expressions subtype predicates are added which can mess up the subexpression paths. We made the translation more robust by exploiting the fact that these predicates are always conjoined at the beginning of a sub-formula. We made changes to our translator from Specware logic to Isabelle logic to reflect the use of named predicate subtypes – previously the translation was losing the predicate subtype, thereby some proofs to fail.

Another difference/change: support type refinement such as occurs during the finalizeCotype transformation where a previously abstract type is refined to be a record type. Isabelle requires that type symbols and their definitions be introduced at the same time, which a refinement system like Specware does not. To support this we introduced a transformation to explicate the previously-implicit morphism that arises when a type symbol is defined later than its introduction in a Specware spec. The morphism is between the spec with the abstract type and the spec with the defined type. The obligations of the morphism are that the axioms on the abstract type are theorems on the defined type. In the case of finalizeCotype, the relevant axioms are that the post-conditions of the state transformers are true given their preconditions. These pre- and post-conditions are preserved in the final spec so the obligations are trivially true. The finalizeCotype transformation also provides bodies for the functions specified by pre- and post-conditions, so we also have the obligation that the bodies satisfy post-conditions given the pre-conditions, which follows simply given that the bodies are mechanically derived from the post-conditions.

Another extension: The Specware rewrite engine includes some built-in speculative transformations such as expanding let bindings and pushing functions inside if-expressions that may enable the application of the main transformations. We extended the proof-emission capability to take account

We also revised our approach to a key problem in generating Isabelle proof scripts. The problem has to do with the straightforward notion of substitutivity:

if x=y then f(x)=f(y)

When we transform an expression f(x) by simplifying its subterm x to y, then we want a proof that f(x)=f(y). The problem has been identifying to Isabelle which subterms x and y are equal, since the paths to the subterms are typically modified during our translation from Specware to Isabelle. Previously we had been explicitly giving the context/path to x by means of a lambda

lambda(v)f[v]

to indicate the hole where the subterm x occurs. We found a simpler solution in using the argCong mechanism of Isabelle, which automatically searches for the subterms x and y and then infers the desired result f(x)=f(y). This helps in pushing through the proofs emitted by our finalizeCotype transformation.

Another problem has to do with the handling of conditional rewrites. In addition to conveying the condition of the rewrite to Isabelle, sometimes the variables are quantified over a subtype, so the subtype effectively becomes an additional condition. We extended our translation mechanisms accordingly.

Another problem arises due to the use of speculative rewriting in the rewrite engine. Some rewrites may not improve the code so they are applied speculatively, and if they do not enable an improvement, then they are withdrawn and rewriting continues. Obviously we do not want that backtracking reflected in the generated proof structure, so we added a mechanism to detect backtracking and to produce a proof script reflecting the actual path to the transformed results.

Several other improvements to our translation from Specware/Metaslang to Isabelle. First, the translate construct is used to rename symbols from an imported theory. For example, the theory of linear orders might have its type renamed `time' in order to provide a simple appropriately named theory of time. The translate construct though caused an exponential blowup of copying in our previous implementation, so we needed to cache translated imports to avoid duplication. This problem only arose as we introduced a monad for formalizing the interleaving of threads that we need to specify and reason about the concurrent execution of mutator and collector. We also fixed errors in our spec-substitution construct, which was causing problems in translating proof terms from Metaslang to Isabelle. The solution was to apply substitutions to specs but not the spec-element terms, but instead to regenerate them, exploiting context.

We continued to develop and store proofs with theorems for the specs in the Specware library. The derivations invoke theorems from imported specs to perform rewrites and the generated Isabelle proof scripts depend on those library proofs.

3.8.3 Locales for capturing proofs of library refinements

Some refinements are generated by composing a library refinement with an application-domain specification by a pushout operation (realized by Specware's substitute operator). In this case the proofs of the library refinement are expressed in terms of the domain and codomain symbols, but we want a proof in terms of the application-domain specification. In other words, we needed a uniform way to generate proofs when the refinement works by pushout/substitute rather than by the specification-specific calculations that other transformations use. One approach is to use Isabelle's locale mechanism which allows a kind of generic proof that can be instantiated in a way that mirrors a specification pushout/substitution. Rather than include the details, we refer the reader to the technical note [Kreitz12] which shows how to develop a locale for the fixpoint iteration algorithm theory discussed above.

3.8.4 Proof Script Generation

We designed a mechanism to augment the transformations used in the CGC derivations so that they both generate a refinement and emit proofs. The ISAR interface to Isabelle provides a format for calculational proofs that clearly reflect the equational reasoning carried out by our transformations. To illustrate, if we perform a calculation of the form

```
A = B       by rule r1
  = C       by rule r2
  = D       by rule r3
so A=D.
```

then we can generate an Isar/Isabelle proof script of the form (where some detail is elided)

```
proof
  have          "A
                     =  B"        by (… rule r1 …)
  also have     "... =  C"        by (… rule r2 …)
  also have     "... =  D"        by (… rule r3 …)
  finally have : "(A = D)" .
qed
```

This proof script can then be automatically checked by Isabelle. This means that an external certifying authority need not trust our transformations. Instead, we will be able to generate both code and proof (i.e. proof-carrying code), and let the certifiers have the mathematical evidence that the code meets its requirements, which they can check,

independently and cheaply. We have some simple cases working now and plan to expand coverage to include most, if not all, of the transformations used in the derivation of our collectors. This process will be spread over several months, since each transformation must be augmented with code to emit proof scripts in the above form. A parallel effort is also required to build up the proofs for basic theories in Isabelle and discharge the theorem in our domain specifications; e.g. we must have a proof of rules r1, r2, and r3 in the example (which is trivial if they are axioms).

To be clear, Specware's transformations automatically carry out the calculation, and our objective for the next month is to have them also automatically generate the corresponding Isar proof script. This means that each generated refinement also has a generated proof that discharges its obligations, without having to perform a post-hoc proof search. The Isar proof script is formulated to put Isabelle on a very tight leash – its proof steps are tightly controlled, so that it will not get in trouble by attempting to search. After all, the transformation knows the structure of the calculation, so that is reflected in the proof script. We believe that this approach to proof generation will be dramatically more economical than post-hoc verification.

We worked on a general mechanism for generating proof scripts (expressed in the ISAR dialect of Isabelle) from a derivation script. Since most of our transformations work by chaining equations, we worked to capture this form of calculation in a generic way. Some of the details include the need to specify which subterms to perform matching on so that Isabelle as a proof checker doesn't need to search, and the need to relate the results of equational calculation to the implicational proof obligations that are generated for refinement steps in Specware. Since some of the transformation steps involve inequalities (versus equations), a next step is to find a way to output proof scripts with implication chains rather than equation chains; i.e. $A \Rightarrow B \Rightarrow C \Rightarrow D$, so $A \Rightarrow D$, rather than the equational chain $a=b=c=d$, so $a=d$.

We streamlined the presentation of the proofs in ISAR. A typical calculation is focused on a subexpression s of a function or axiom e, so the proof should be presented mainly at the level of s rather than e. The calculated change to s is finally shown to result in the desired change to e. One problem with emitting proof scripts during specification transformation is that Isabelle doesn't support type symbols that are introduced but not defined. It is a recurring problem that Isabelle wasn't built to support refinement processes. We worked on this issue.

We converted over to using Isabelle 2013, which now requires coercions between sets and predicates (previously they were equal rather than isomorphic). This entailed change to the proofs of our base libraries as well as changes to our translator from the

MetaSlang logic to Isabelle's logic. We continued to work on coalescing variant theories in our library, and introduced new specs for bounded natural numbers and integers. This allows us to specify, say, Nat16 for 16-bit natural numbers, which translates to uint16 in C. Another aspect of this task is maintaining proofs for all theorems in the library specifications. The change to Isabelle 2013 entailed some work to re-establish theorems, mainly by modifying some theorem's proof tactic. We found that there were about 10-15 theorems per specification that needed to be reproved.

We also began work on a new abstract syntax for recording proof information at transformation-time. The goal is to have all transformations in our library record the calculations and decisions that they make in this proof structure. We would then be able to uniformly translate from this proof structure (in MetaSlang) into the proof language of a proof checker (Isabelle for now). We extended our proof language to include more information in proof objects, such as errors and context. We made the actual theorem to be proved more explicit in the proof object, which was needed to support better proof combinators.

3.9 Specware Infrastructure

We extended Specware's infrastructure in a number of direction to support the coalgebraic specifications and their refinement.

We improved Specware's transformation for Isomorphic Type-refinement, so that it handles patterns. We improved the rewriter's handling of curried functions. We modified Specware's type-checker algorithm to generate type-coercions, which means fewer proof obligations are generated. We improved printing of Specware specs and terms, and improved the efficiency of code generation for both in time and space usage. We extended the type-checker so it could infer tighter sub-types for the results of ops with specialized inputs. We added indirection construct to pragma language to allow proofs to be separated from specs, so the specs are more readable. We improved the proof obligation generated for a refined op so that it is easier to prove – making the obligation extensional and including subtype conditions of argument variables.

We implemented a version of function unfolding that works with functions specified using pre and post-conditions, by combining the postconditions. We also adapted the common expression abstraction tactic to work properly with assignment statements.

3.9.1 Higher-Order Matching Algorithm

We implemented a feature in the higher-order matcher where it avoids generating subgoals for a subtype mismatch that could be discharged by subtype obligations. Previously, if the term being matched had an associated subtype obligation, to show the rule matched, one would have to prove the obligation was true using the rewriter. This was at best inconvenient. Now we assume that obligations are proved in Isabelle.

We fixed type matching in Specware's higher-order matcher – a type variable is now bound to the least supertype of all the types it is matched against. We also made changes to the Isabelle translator since, in some cases, it was not extracting composite subtype predicates correctly for nested subtypes. We also needed to rationalize the ordering of the extracted predicates.

3.9.2 Support for calculation

Support for calculational inference was extended from equational to handle conditional equations and to handle strengthening of propositions (e.g. the Observer Refinement calculation above).

3.9.3 Tactic language

We also modified the transformation script language to make it simpler to read, write, parse, and automatically generate scripts. This has allowed us to reformulate several existing transformations into the following normal form: generate a derivation script and then run it. This normal form has several advantages:

1. it replaces the writing of arbitrary metaprograms that manipulate abstract syntax,
2. it extends the range of people who can write transformations
3. it prepares the ground for emitting proofs as a by-product of transformation.

We extended the scripting language to support verbatim text for generating into CommonLisp. This allows us to add Lisp-specific instrumentation, monitoring, and other support code as an integral part of the derivation script. We fixed the error handling for transformation moves that fail, so an error message is presented instead of going into the debugger.

3.9.4 Transformation Support

We improved Specware's transformation language machinery so that it is easier to add new transformations both for spec-level transformations and term-level transformations. Now, it is only necessary to define the transformation as a function with a suitable type, without having to add special interface code to the transformation engine.

We added support for user-defined transformations. Previously, adding a new spec transformation function would involve changes to the transformation language parser. We have now implemented a scheme whereby the signature of the transformation function determines the syntax in the transformation language. This makes it much easier for developers to incorporate new transformations into the transformation language, especially when the transformation has multiple options and lists of rewrite rules or functions as arguments. To implement this interpreter capability we had to augment the code generator to output type information for transformation functions so its arguments could be interpreted at run-time. As the interpreter has to work with objects of multiple types, we needed to tag values with their type and provide an interface to the transformation functions that accepts these tagged values.

The basis of the transformation system extension is to have the signature of the Specware transformation function determine the syntax of its use in the transformation language. For example,

```
op MSTermTransform.rewrite: Spec -> PathTerm -> RuleSpecs
                                    -> RewriteOptions -> MSTerm
type RewriteOptions =
  {trace    : Nat,        % Trace level 0, 1, 2, 3
   debug? : Bool,         % Debug matching of rules
   depth    : Nat}        % # of rewrites allowed
```

is the (slightly simplified) signature of a rewrite transformation that transforms the current term using a list of transformation rules and with three options. The "MSTermTransform." qualifier tells Specware that this is a term transformation. The spec and the term are implicit, i.e. given by the current transformation context. The syntax for using this in a transformation sequence is, for example:

```
rewrite [unfold open?, lr mapFrom_TMApply, lr filter_true]
      {trace = 2, debug?= true, depth = 5}
```

where *unfold open?* is the rule for unfolding the definition of open? and *lr thm* takes an equality theorem thm as a left-to-right transformation rule. The system allows for defaults everywhere so the options between braces can usually be completely omitted, or any subset can be specified. E.g.

```
rewrite [unfold open?, lr mapFrom_TMApply, lr filter_true]
```

or

```
rewrite [unfold open?, lr mapFrom_TMApply, lr filter_true] {depth = 5}
```

or just

```
rewrite
```

which just uses the built-in simplification included in the rewriter without any rewrite rules.

Previously, allowing all these syntactic options had to be specifically programmed, so changing an interface, in particular adding options, was a significant amount of work that required knowledge of the internals of the syntax system. Having the syntax automatically follow from the signature makes it easy for any Specware user to add new transformations or extend existing ones.

3.9.5 Tracing support

We also made improvements to the transformation system so that it prints out a much better focused presentation of its (mostly) equational calculations.

3.9.6 Specware Library

We coalesced several variants of specifications for finite sets, bags, lists, maps, stacks, as well as standard refinements of them. We extended the Specware specification libraries with more proofs of theorems, which are used to support calculations at program-synthesis time.

We improved the Specware DataStructures library, with an emphasis on pushing the proofs through Isabelle and fixing any issues revealed in the process. The DataStructures library defines and refines container data structures, including Sets, Bags, Maps, etc. We added many Isabelle proofs (including proving quite a few new, generally useful auxiliary properties). Perhaps the most interesting proofs were those justifying the correctness of the refinements (expressed as morphisms) of various structures in terms of the others, many of which make heavy use of 'fold' operations.

The library work is still ongoing, but the Sets and Bags libraries are now completely proven.

We also refurbished many of the specs in the Crash repository, to get them working with the latest version of Specware and its libraries. We also worked on Specware documentation, testing, and miscellaneous maintenance tasks and improvements (e.g., modernizing the syntax of important specs).

3.10 Generator of imperative code

We looked at the problem of generating imperative code from monadic code using a state monad, and studied the literature on this subject, particularly in the context of Haskell. We created a prototype implementation of one approach that involves unfolding all the monadic functions and simplifying the result to get rid of their overhead and hand-implementing some low-level functions as assignments. This was largely successful, with some further work being necessary to remove some remaining overhead. This transformation required some minor extension to the matching component of the transformation system to handle a sequential composition pseudo-function.

However, we are also exploring an alternate approach to achieve the same end. Our analysis suggests that it is may be more straightforward to perform single-threadedness analysis on the low-level design and then translate the specifications directly to imperative code.

Towards a C generator, we are constructing a sequence of specification transformations that correspond to compiler passes and that are intended to be simple enough that we can augment them to emit proofs at application time. We completed transformations for linearizing nested terms in single-threaded state transformer definitions and related code needed to prepare for globalizing the single-threaded state in our coalgebraic operations. We made numerous other internal improvements. We worked on issues related to handling pattern-matching in the compiler – since C doesn't support patterns for de/construction, there is no direct translation of this feature of MetaSlang, so special control mechanisms are needed to handle matches that partially succeed before failure.

We extended earlier work to propagate type information through our abstract syntax trees so that ambiguous constants (such as 1) can be consistently typed when passed to C. We made many internal improvements in support of C generation. We worked to generalize and clean up the transformation sequence that generates C (about 23 transformations), and to develop a compilation specification that allows expressing some C-specific information: import files, native library types and functions that are used in the MetaSlang specification, translation of field names, and any special-case

definitions. During this period we were able to generate, compile, and run idiomatic C code on some sample specifications written in our mixed algebraic/coalgebraic style.

3.10.1 Language Morphisms

Language morphisms are a generalization and formalization of what had been ad-hoc features for translation to Isabelle and Haskell. Special ``translate'' pragmas within a spec can now be used to define language-specific rules for translating Specware types and ops. These pragmas now have an internal structure that is parsed in a very generic manner to obtain five kinds of information:

3.10.1.1 Imports

This section simply lists a sequence of files to be imported into the generated target file. For example, a translation to C might include:

```
-import
  stdlib.h    % boilerplate
  linux/udp.h % structures specific to UDP protocol
  mycode.h    % interfaces to ad-hoc application-specific code
```

3.10.1.2 Verbatim

This section is intended to be used sparingly, but provides an escape mechanism to insert arbitrary text verbatim into the target file. It is intended to handle ad-hoc problems that resist a generic solution.

For example, the * operator in C is a function and can be modeled relatively simply within Specware, but & is not a function since substitution of equals for equals fails. There thus is no simple way to target C expressions headed by &, but some special cases can be handled on an ad-hoc basis by allowing Specware operators to map to C macros that include &. For example:

```
-verbatim
  #define atomic_read_at(x) (atomic_read(&x))
```

Verbatim text may create problems for verification, but it isolates such problems to a small set of clearly identified operators.

Also, because such verbatim text must appear declaratively within the specs being used, those that lack such tricks can be known to be free of such problems — there is no programmatic mechanism secretly including such tricks as part of the translation.

3.10.1.3 Translate

This is the main section, and provides for translation of specware names (for types, ops, and field references) to target names or terms, along with an indication as to the location of the target (primitive/syntactic or file location). Translations to complex terms are implemented as target macros.

```
-translate
  type Nat.Nat32    -> uint32_t      primitive
  op Nat.BVAND32    -> & infix       primitive
  type udp_table    -> struct udp_table      in net/udp.h
  field udp_table.csum  -> udp_table.check   in net/udp.h
  op Null_ID      -> ((Sock_ID) NULL)    macro
  op sizeof_udp_hdr   -> sizeof (struct udphdr)  macro
```

3.10.1.4 Native

This section provides a simpler form of translation where the named type or op is assumed to translate directly to the same name in the target.

```
-native
 op ntohs in /drivers/staging/rtl8712/generic.h
 op udp_hdr in linux/udp.h
```

Language morphism pragmas for any given spec are collected recursively through all imported specs, making it possible to distribute the language-specific translation rules for types and ops into the local contexts where they are introduced or defined. Alternatively, the translations could be handled en-masse by one pragma in the top-level spec, for example if one wished to have alternative top-level specs with different rules targetting different compilers.

Future work could easily validate that a type or op declared to be in a target file was at least nominally present there. With language-specific parsing of the target files. it would be possible to verify appropriate typing, etc.

3.10.1.5 Slices

A perennial problem with processing specs has been that each processing context may be concerned with just some aspects of a spec, requiring ad-hoc code to determine which elements of the spec to process and which elements to ignore, making such processing fragile and hard to maintain as Specware evolves.

One aspect of this problem is that alternative notions such as defined, executable, implemented, primitive, hand-coded, etc. have tended to be conflated within such processing code, sometimes confusingly (and even inappropriately) using the same tests in contexts where slightly differing ones were needed.

There also were early attempts to create more manageable artifacts by simply subtracting out undesired portions of a spec, however this led to ill-formed specs that contained the information of interest but were missing semantically important theorems, subtype predicates, etc.

Slices provide progress towards a generic solution to this problem by layering filters over specs to provide ad-hoc tailored views. They leave the spec itself unaltered but add tables describing which elements of the spec have various desired attributes. Each particular processing context can then view the spec through such a filter, simplifying the processing context while avoiding logical problems associated with ill-formed specs.

3.10.2 Concurrency Support

We implemented a new AddMutexes transform that is restricted to generating mutexes for primitive transformers (procedures) that affect an invariant – the effect of the mutex is to ensure that the invariant is never observed to be violated. Our intention is to first generate coarse-grain mutexes that produce large but correct atomic regions, then to introduce concurrency-improving transformations that break large-grain regions correctly into finer-grain regions.

We are exploring two approaches to formally representing the top-level specification of a collector and a mutator and proving the safety and progress of their concurrent execution. Challenges arise due to the natural interference of each component with the other, and their concurrent execution. To model the interference and relative noninterference we use rely conditions in the form of transition invariants that each component assumes about its environment (i.e. the other component). The Collector assumes that dead nodes are increasing monotonically by the action of the environment, and the Mutator relies on the invariant that the live nodes on the heap are isomorphic from moment to moment.

One approach to formalizing these rely conditions and to proving safety and progress was presented earlier in Section 3. A second approach is based on a monadic formulation of the mutator and collector as state machines whose execution steps correspond to atomic actions and whose execution can be given an operational (vs denotational) semantics via interleaving of the atomic steps. This approach allows us to specify the system architecture as the concurrent execution of two state machines, while allowing our existing derivations in Specware to generate the low-level code for the atomic steps. The monadic structure provides the control structure and interleaving

of execution steps. Our ongoing efforts are (a) to reconcile the elegant proofs afforded by the algebraic approach, with the more automatable monadic approach, and (b) to complete the specification of the system and link it to our several derivations of garbage collection algorithms.

We continued to develop and store proofs with theorems for the specs in the Specware library. The derivations invoke theorems from imported specs to perform rewrites and the generated Isabelle proof scripts depend on those library proofs. In several instances we found the need to add conditions to theorems to enable proofs. This then requires ensuring that the mutator and collector operations have pre/post-conditions strong enough to discharge those new conditions.

3.11 Flex Seedling

3.11.1 Executable Prototype

In order to run tests and experiments early in the project, we started by writing an executable specification of a simple resolution theorem prover. This simple theorem employs a saturation algorithm that exhaustively applies binary resolution and factoring to the input set of clauses until (1) a contradiction is derived (in which case the original conjecture is proved), or (2) the set of clauses is found to be satisfiable (in which case the original conjecture is not provable), or (3) a resource limit (provided as input) is reached (in which case the original conjecture may be provable or not).

As we started testing this simple prover, we confirmed the need, in order to run sufficiently interesting examples, to extend the simple prover with the following features, which are quite standard in resolution theorem provers:

- **Tautology Removal.** A tautology is a clause that includes a literal and its negation. Tautologies are always true and do not contribute to the proof goal, and can therefore be safely eliminated as soon as they are generated.
- **Subsumption.** A clause subsumes another one when the former is more general than the latter, i.e. the latter can be directly derived from the former. Subsumed clauses do not contribute to the proof goal and can therefore be safely eliminated as soon as they are generated – this is forward subsumption, i.e. when a newly generated clause is subsumed by an old one. Backward subsumption occurs when a newly generated clause subsumes an old clause: in this case, the old clause can be safely eliminated while the new clause is retained.

- **Set of Support.** Set-of-support is a technique to restrict binary resolution to only resolve two clauses picked from two different sets: (1) the set of hypotheses; and (2) the set consisting of the negated conclusion and of the resolvents generated so far. The latter is the 'set of support'. The rationale is that the hypotheses are expected to be consistent, but contradictory with the negated conclusion: thus, resolving clauses from the hypotheses should not contribute to the proof goal; the contradiction must involve the negated conclusion and its descendants.

- **Demodulation.** Demodulation is a technique to handle equality formulas efficiently. In principle, equality can be handled by adding appropriate equality axioms to the hypotheses, but this is generally inefficient because of all the congruence axioms needed for the functions and predicates that appear in the hypotheses (given that a resolution prover operates on first-order logic). With demodulation, equality singleton clauses are used as rewrite rules for terms.

3.11.2 Pre-Filter Optimization

With the executable Flex prototype in hand, we proceeded to try and apply various optimizing transformations to the prototype. All the transformations operated automatically, but they were manually applied, i.e. we chose which transformations to apply and the parameters to supply to each transformation application. This was an exploratory activity, before building facilities to automatically choose the transformations to apply and the parameters they apply.

As first target for our optimizations, we looked at the unification algorithm of Flex. In resolution theorem provers, unification is a fundamental procedure that is used as part of binary resolution: candidate literals to be resolved are unified before being resolved. Since unification is one of the most heavily used procedures in a resolution theorem prover, it is a good candidate for optimization.

A pre-filter is a necessary condition for the successful evaluation of some other, more expensive condition. In this case, the unification operation ("is there a substitution that makes two terms equivalent?") can be expensive to compute because it involves creating possible substitutions and checking consistency of variable assignments. A fast pre-filter can test whether the skeletons (e.g., the trees of function calls and constants that appear in the terms, ignoring variables for the moment) of two terms are such that they might unify. If not, there is no reason to spend the time building up substitutions, because the unification will ultimately fail. Such strengthening or weakening is important for other optimizations such as deriving pruning tests in search algorithms.

We explored several versions of a pre-filter for the unification algorithm. In doing this we tried to simulate what an automatic search algorithm could do when looking for

optimizations of this kind. Running test cases showed that some of these generated pre-filters could lead to significant speed-ups for certain unification problems, whereas in others the speed-up was less than the added overhead. From this exploration it does appear that pre-filter optimizations could be automatically derived and proved using a generate-and-test approach with a fairly naive generation strategy.

We derived a fast-fail pre-filter for unification, using our transformation system. The pre-filter allows the operation to fail quickly in the common case where the terms being unified do not match. In particular, it causes unification to fail if the structure of the terms is such that they could not possibly match, and it does this quickly, without bothering to build variable substitutions and check them for consistency. If the attempted unification passes the pre-filter, the full unification algorithm is invoked.

Our derivation of the fast-fail pre-filter for unification consists of a sequence of 6 automated transformations: expand-lets, wrap-branches, simplify-body, strengthen, drop-function-from-nest, and drop-irrelevant-parameters. Each invocation of a transformation is short (often just a single line) and produces a new mutually-recursive set of predicates (usually three predicates: for attempting to unify a term, a variable, and a list of terms). A typical transform is implemented in a few hundred lines of code and generates a few hundred lines and code and functional correctness proofs. The sequence of 6 transformations automatically generates an optimized unification algorithm that includes a fast-fail pre-filter, along with a proof of functional equivalence of the optimized version with respect to the original version of the unification algorithm (i.e. the version without the fast-fail pre-filter optimization).

3.11.3 Finite Differencing

Finite differencing, also known as incrementalization, is a well-known program transformation technique in which results from a previous iteration (loop or recursion) are cached in a way that accessing their cached values, and updating the cached values at each iteration, is faster than computing them from scratch.

We implemented a general-purpose finite differencing transformation in our system, and we applied it to the following two procedures that are part of the Flex prover:

1. The extraction of demodulators from the current set of clauses (this is part of demodulation, described earlier).

2. The calculation of all resolvents from the cross-product of the current set of clauses.

As part of this process, we also proved theorems that the finite differencing uses to simplify the updating expressions, e.g. distributive properties of the operators involved.

In the process of applying these finite differencing transformations, we found that the representation of clause sets as lists made it difficult to use suitable distributivity laws to optimize the computation (i.e. extract demodulators and calculate resolvents). Thus, we changed Flex to use a higher-level, more abstract representation of clause sets in terms of mathematical sets, using an existing library for sets.

We also investigated the following topic. After applying finite differencing to an iterative computation, often the code maintains the cached information at every iteration, including the last iteration, which is often unnecessary. In order to eliminate this inefficiency at the last iteration, we developed a new transformation to restructure a "while-like" loop (which performs the test at the beginning of each iteration) into a "do-while-like" loop (which performs the test at the end of each iteration). This results in a little code duplication, but enables finite differencing to avoid the maintenance of the cached information at the last iteration, resulting in time improvements.

3.11.4 Declarative Specification and Formal Refinements

After experimenting with optimizing and testing the executable Flex specification, we developed a declarative specification of Flex, with the intent that the executable one be a refinement of the declarative one. The declarative specification does not include subsumption, set of support, and similar features. Instead, it includes under-specified components that can be instantiated to add those features. In particular, the declarative specification includes an underspecified filter that allows any clause to be dropped (which is always sound): by suitably instantiating that filter, we can remove tautological clauses and/or subsumed clauses.

We have developed and proved formal refinements from this high-level declarative specification of Flex to versions that incorporate domain-specific optimizations like subsumption, tautology removal, and set-of-support. These specifications and refinements form the Flex derivation tree. The executable Flex specification described earlier can be connected via formal refinements, to this derivation tree.

3.11.5 Testing

To test Flex, we drew tests from the following sources:

- Tests that we built specifically to test certain features.
- Arithmetic and logic puzzles taken from [Pelletier86].
- Tests from [AAR15].
- "Thousands of Problems for Theorem Provers" [TPTP], a large library of theorem proving problems and proofs, covering a wide range of difficulty, topics, provers,

and logics. It is the standard test suite used in the annual automated theorem prover competition at CADE.

We drew most of the tests from TPTP. Since those tests are written in their own TPTP format, we developed a translator from the TPTP format to the Flex format, based on existing translators for other theorem provers that come with TPTP.

Figure 7: TPTP testing

We developed an automated test harness that runs the tests on all the executable versions of Flex, collecting information such as success/failure, times, number of clauses processed, and so on.

As part of this test harness, we developed a graphical interface, shown in Figure 7, to display the Flex derivation tree and the results of running each executable version of

Flex on the test suite. The display is updated in real time as the tests run. The interface includes bar charts to compare the results of the different versions of Flex. The purpose of the test harness is (1) for Flex to run it autonomously to collect empirical information that will be used to steer the development of Flex and of its optimizations, and (2) to use the information collected by the test harness to generate readable tables/reports.

4 RESULTS AND DISCUSSION

Figure 8 shows a roadmap of the derivations we performed in this project, with completed derivations shown in red boxes. Each derivation starts from a common specification of the Collector. The Reference Count collectors are mainly derived via Observer Maintenance. The Copying, Generational, and Marking Collectors all stem from a fixpoint algorithm for tracing live nodes, but differ in their memory model and many other details. In the following sections, we give more detail of each derivation.

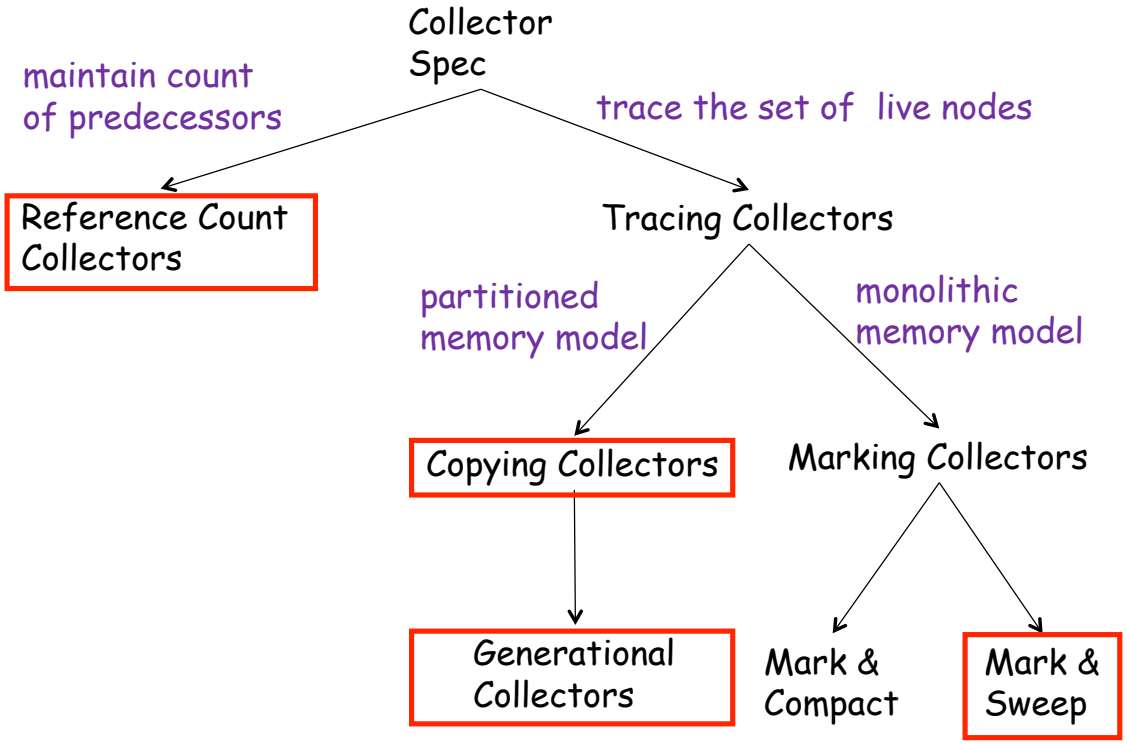


Figure 8: Derivational Family Tree of Collectors

4.1 Generating a Concurrent Mark&Sweep Collector

Figure 9 is a summary of the sequence of transformations in the derivation. The Mark&Sweep derivation starts with the application of the fixpoint iteration algorithm theory to generate high-level algorithm for tracing the live nodes. Since the algorithm theory introduces the Workset observer, it naturally follows to apply Observer Maintenance on WS. At this point the Graph cotype has been renamed to Heap and then to Memory. The next eleven steps are optimization transformations that refine abstract observers, simplify expressions, and introduce new observers to speed up computation (e.g. rootCount which maintains a count of the number of current roots).

C1. Algorithm Design
C2. Simplification
OM1. Observer Maintenance: WS
Mem. rename {Heap +-> Memory}
OR0. Observer Refinement of payload
OR1. Observer Refinement: tgt →tgtIM
OR1a. Observer Refinement: outNodes →outNodesIM
OR2. Observer Refinement: roots → rootsL
OM2. Observer Maintenance: rootCount
OR3. Observer Refinement: nodes → nodesPair
Mut1. Import random mutator
Mut2. Simplify
OR4. Observer Refinement: supply → supplyL
OM3. Observer Maintenance: supplyCount
OR5. Observer Refinement: black → blackCM
OR6. Observer Refinement: WS → WL → WStack
Cot1. FinalizeCoType Memory
Cot2. Define initBlackCM, …
Iso1. Type Isomorphism: Memory ⇔ Memory'
DTR1. DataType Refinement: Maps → Vectors
DTR2. DataType Refinement: Stacks → Vectors
DTR3. DataType Refinement: Sets → Lists
G1. Globalize Memory
D. Simplifications
Cgen. Code Generation

Figure 9: Derivation Structure for a Mark&Sweep Algorithm

At that point, the abstract observers are sufficiently refined that we can gather them and define the Memory cotype via finalizeCotype, which also synthesizes definitions for any transformers that were specified by undefined. The type isomorphism transformation is used to package three observers into one: black (marking bit), payload (data), and tgtIM (outgoing arcs) become fields of a heap cell. An alternate way to organize memory is to have a separate marking array and package payload and tgtIM – this would be accomplished via a different application of the type isomorphism transformation. The derivation then performs three datatype refinement steps using library refinements to implement Maps, Stacks, and Sets. Finally, the Globalization transformation (1) introduces a global variable for Memory that comprises the entire address space, (2) implements the single-threaded transformers via side-effecting operations on the global Memory, and (3) introduces mutexes for the bodies of atomic transformers in the case of a concurrent collector. Lastly, the code is turned over to a conventional compiler to produce the binary.

The derivation in Figure 9 was the result of extended exploration – we have recorded some 25 folders containing variants of the domain theory and derivation for Mark&Sweep alone. We list a few of the issues that motivated the search:

1. Formulating the domain theory and problem specification. For example, we spent several versions exploring the use of sets versus bags, collections (a weaker version of sets) , or Lists to model arcs. Various problems of refinement motivated alternative formulations, until we discovered the Observer Refinement transformation, which allowed us to cleanly use sets (the most natural formulation).

2. How to handle references – after many alternatives, we settled on unique Identifiers as the correct abstraction rather than addresses (which is just one implementation of an identifier) or polymorphic pointer types.

3. Formulating the Fixpoint Algorithm Theory – We developed many variants, including functional, state-based, and concurrent versions.

4. Organizing Derivations – The GC derivations are complex enough that we needed to develop techniques for organizing and managing them. For example, we used a spreadsheet to track the status of observers: introduced, defined, maintained, ghost, or refined. At each refinement level, this helped to track which observer postconditions needed to be added when a transformer is introduced. For another example, we learned to factor refinements into definitional extensions to aid in composition during refinement. We also developed a version control strategy of creating new folders for all derivation information when starting a new approach (hence the 25 versions mentioned for Mark&Sweep).

5. Performance Issues – Performance profiling motivated many reformulations and transformations. A simple example is the size of the supply list. Some code required the size to determine if the collector was thrashing. The Observer Maintenance transformation was then applied by introducing a new observer, supplyLength, with the invariant

   supplyLength st = length (supply st)

   where st is the state. An open issue in program synthesis is how, in general, to guide the derivation process to achieve performance (or other nonfunctional) goals.

One other performance improvement required deeper insight and changes. As noted in an earlier status report, one general lesson about coalgebraic specification and refinement is the need to distinguish *identity* and *value* of the elements of a coalgebraic type (e.g. the state). This is not a distinction that arises by refinement, but must appear in formalizing the application domain specification. For heap nodes, this means that we specify their identity (which typically refines to addresses) and a means for accessing their current value via their identity. In the past month we realized that this distinction must also apply to the arcs/pointers as well. At the outset of this project, our concept was that the heap was specified as a graph where we have a basic observation of the set of arcs coming out of a node. However, the set structure doesn't allow the arcs to acquire identities by refinement. The specification must start with the arcs having an identity (which refines to address and offset), together with identity-based access. The set of arcs that go out of a node can be then be computed as an abstraction. This required a more extensive change to the specification and the derivation structure, but allowed much better generated code. It flies in the face of the heuristic to state the initial requirements and domain model in as abstract terms as possible, but we are learning that the identity/value distinction is fundamental for cotypes.

4.2 Generating a Copying Collector

We also developed a derivation of a Cheney-style copying collector. The overall plan was to modify the Mark&Sweep derivation, since both algorithm families are based on an iteration to find the live nodes. Several high-level insights emerged from studying the algorithm and the concepts necessary to specify and derive it.

1. Copying requires a somewhat more general structure than M&S, which is only concerned with finding live (versus dead) nodes, whereas a copying collector has to find and copy arcs/pointers too. We generalized the fixpoint iteration algorithm to find that reachable *graph* rather than just the reachable *nodes.*
2. In exploring the coalgebraic style of specification and refinement, it has become increasingly clear that early on in the specification/derivation process, one must specify the distinction between *identifiers* and their state-based *values*. The identifiers may be names, addresses, indices, etc. with the main requirement being that they uniquely identify some varying quantity. In normal usage, the identifier remains constant and its value may fluctuate with changing state. However, in the case of a copying collector, the converse holds: the identifier is changed and the value remains constant! Underlying a copying collector is a fundamental algorithm for translating identifiers for a collection of values. This algorithm will be based on a building a translation table which is a bijection between old and new identifiers. In copying collectors, the translation table is typically implemented by forwarding pointers. A particular challenge is building a

concurrent version where the identifiers are being simultaneously used by the Mutator and changed by the Collector. The problems of compaction, defragmentation, virtual memory, and network address translation require similar treatment.

3. The safety requirement of a Collector is that its actions should not affect the Mutator's data. Mark&Sweep algorithms achieve this by leaving the graph of live nodes unchanged. However, a copying collector works by changing the identifier/address of live nodes, so it is, in a sense, changing the Mutator's data. A weaker characterization of the safety requirement is needed: the collector preserves the *topology* of the heap rather than its exact structure. *The heap after collection is* ***isomorphic*** *to the heap before collection.* The isomorphism between the before- and after- heaps is exactly the translation table that is built up during copying.

We realized that the crucial invariant of a copying collector is not a state invariant, but a *transition invariant*, which is a property over a pair of states and which is required to hold over all state transitions of a program. The transition invariant for a copying collector is that the graph of live nodes must remain isomorphic under every transition effected by the collector. Note that this is not exactly a conditional invariant, rather is a high-level requirement of any garbage collector. Our previous specification of the requirements on a mark-and-sweep collector had that the graph of live nodes is simply preserved by the collector, which is the special case that the isomorphism is simple equality versus graph isomorphism. The innovative idea we are pursuing is that we can synthesize the core algorithmic parts of a copying collector by means of enforcing the isomorphism as a transition invariant. The driver for the maintenance is the change of identifier (i.e. logical address) that lies at the heart of copying (also for compaction algorithms).

We modified our derivation script for a Mark&Sweep collector to generate a Cheney-style Copying collector. At the algorithmic level, the guiding concept is that a copying collector is a different interpretation of the same abstract design – a fixpoint iteration controlled by a workset. After that, the enforcement that all collector actions maintain isomorphism of the live heap produces key copying actions. After that, there is a similar sequence of optimizing transformations and data structure implementation refinements.

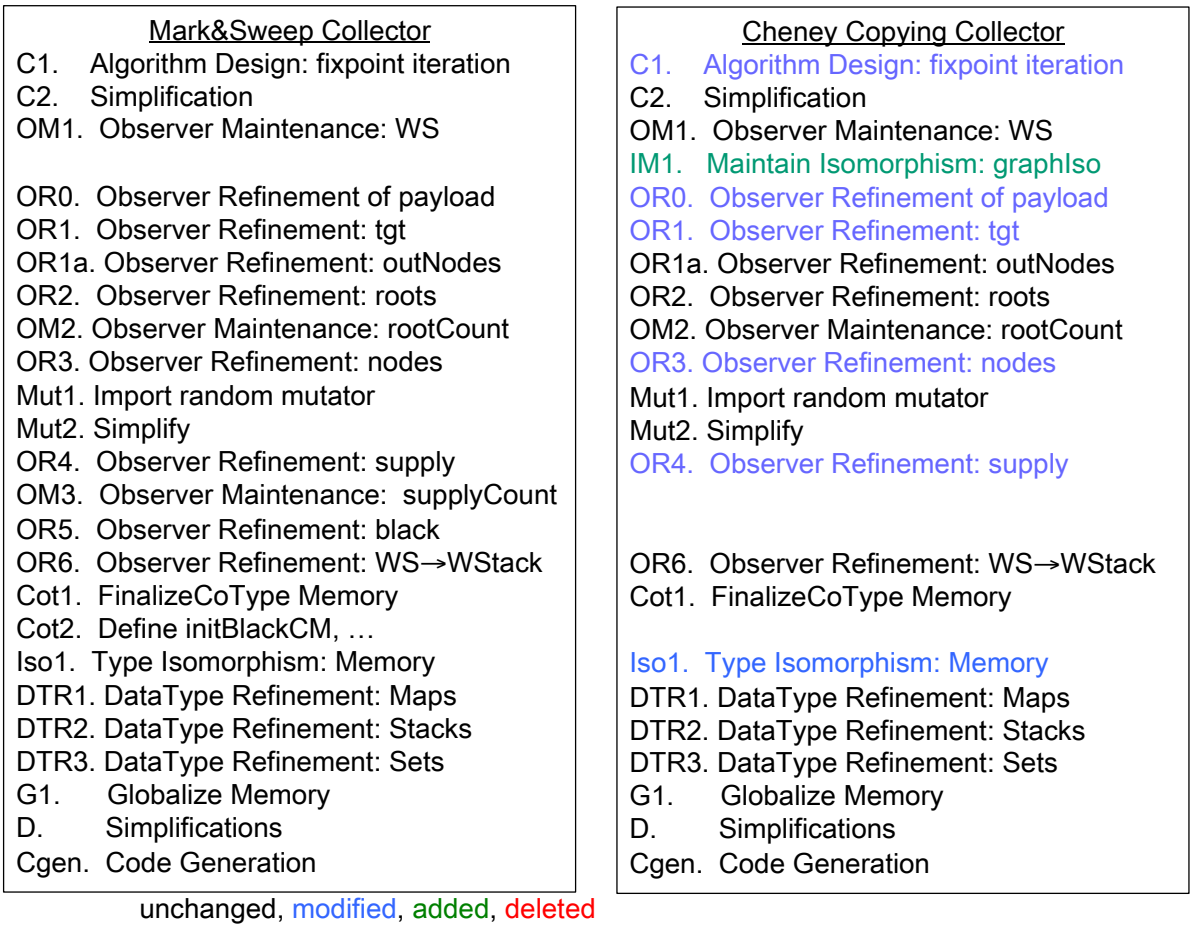


Figure 10: Comparison of M&S and Copying Derivations

A side-by-side comparison of the Mark&Sweep derivation structure with the Copying Collector derivation is shown in Figure 10. Both are comprised of ~25-30 transformations. Of those about half (15) were copied over unchanged to the new derivation: of the 25 steps of the copying collector derivation, 1 was new, 3 were deleted, 15 were used unchanged, and 6 transformation steps were modified to obtain a copying collector. This level of reuse was a little surprising, but it is an expected benefit of a refinement approach that is based on highly reusable transformations.

4.3 Generating a Generational Collector

We generated a simple generational garbage collector by modifying our previous derivation of a Cheney-style copying collector, since a generational collector can be seen as a one-way copying process – rather than alternating between To-space and From-space, a generational collector copies from new space to old space. The technical challenge was to derive the key parts of a generational collector as a result of

enforcing the invariant that the heap space remains invariant under Collector and Mutator operations. We achieved over a 60% reuse of derivation structure between the Copying and the Generational collectors, enabling a significant productivity increase.

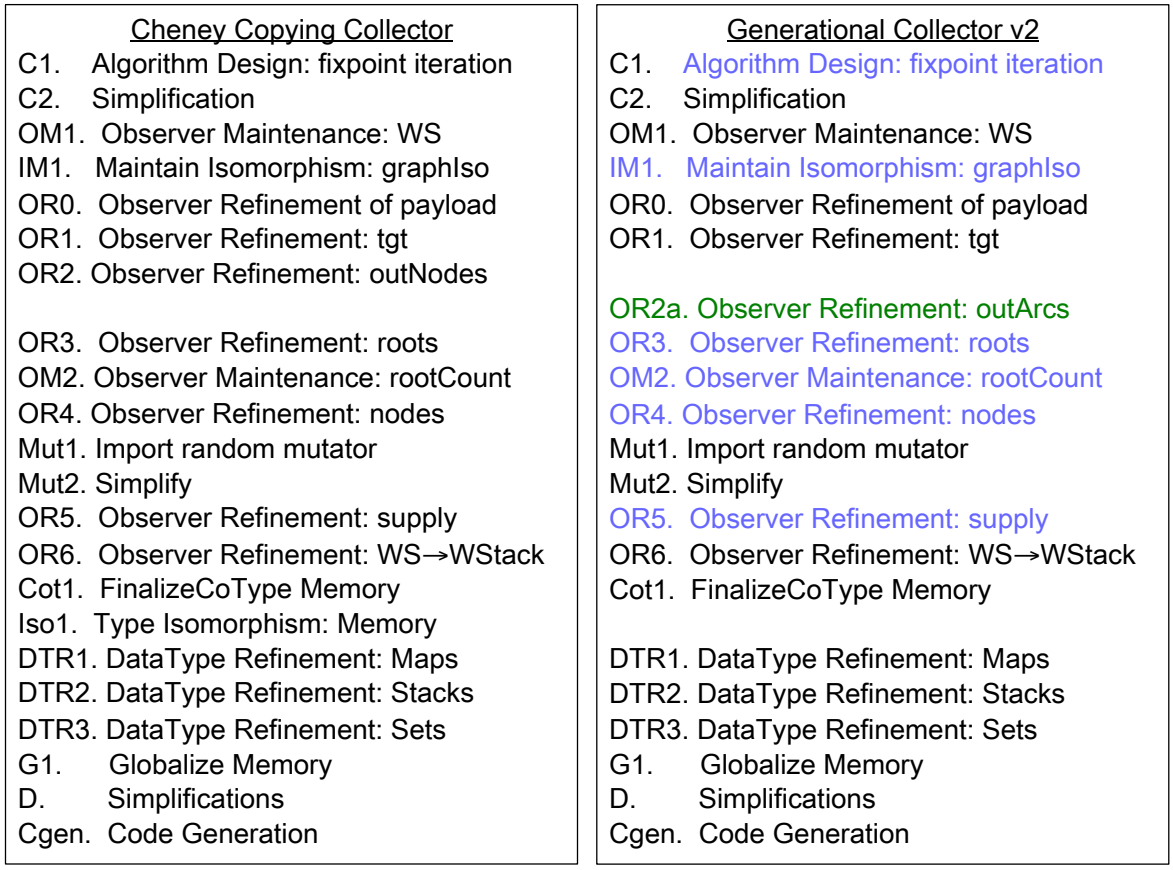


Figure 11: Comparison of Copying and Generational Derivations

4.4 Generating a Reference Count Collector

The reference count on a node n is the number of pointers to n from live nodes. The key idea underlying the derivation of a reference count collector is to maintain the reference-count of a node via Observer Maintenance; i.e. to generate code to maintain the reference count from an invariant. Whenever a pointer is created or changed, the maintenance code updates the reference count simultaneously. It emerged during this derivation that the supply of free nodes could also be maintained via an invariant – the supply is the list of nodes that have reference count of zero. Incremental maintenance will add a node to the supply whenever its reference count is decremented to zero. In this way the "algorithm" of reference count collection emerges from the application of

well-known and highly reusable transformations. A surprising insight from the reference count derivation is that we could go back and revise the other GC derivations to treat the supply observer as maintained from an invariant.

The remaining steps in the derivation are mostly the same as those in the derivation of a Mark&Sweep collector, as shown in Figure 12. The left column shows the steps in a Mark&Sweep derivation/metaprogram. The right column shows the changes from the Mark&Sweep derivation to produce a Reference Count collector. The steps in red are deleted, the steps in green are new, the steps in black are unchanged, and the steps in blue are modified from their counterparts.

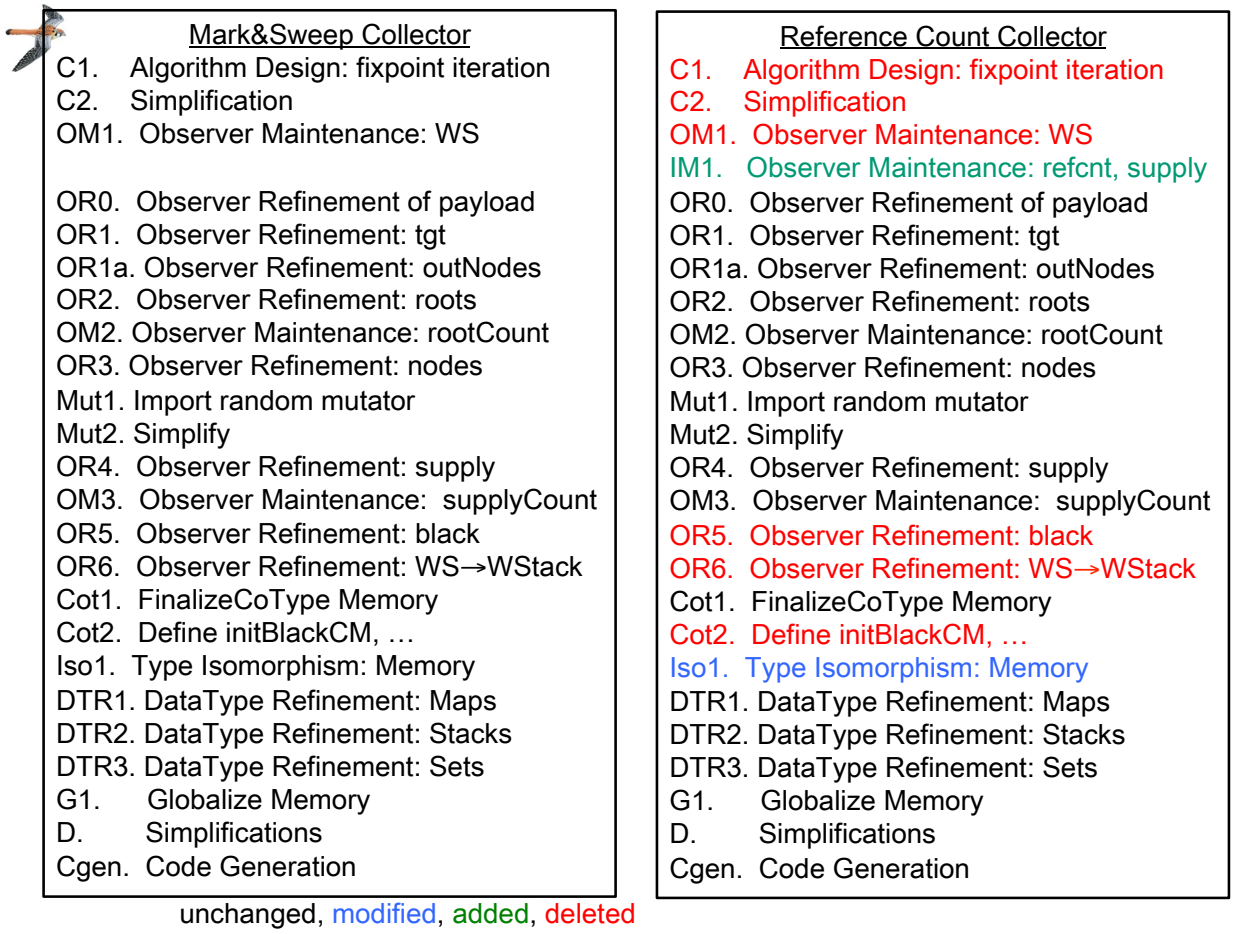


Figure 12: Comparison of M&S and Reference Count Derivations

4.5 Performance Enhancements

We continued our effort to eliminate the generation of garbage by the garbage collector. The change to the graph formulation based on the distinction of identity and value

(reported earlier) enables us to treat this problem. In particular we studied how to preallocate and to reuse memory for the workset (refined to a stack), versus the easier approach of treating the workset as a Lisp list (with its implicit consing).

A more complex example arises from the way we had been performing stateful updates to complex structures, such as the heap itself. In a functional/algebraic setting, a map update typically generates a new map using some newly allocated memory cells and shared structure with the old map. If the old map is never used, then we can reuse old cells rather than allocate new cells and thus avoid generating garbage.

Part of our learning curve has been to eliminate this source of inefficiency by switching from algebraic/functional structures to coalgebraic/imperative structures. We began extending the Globalization Transformation to generate updates to the global state that are maximally localized. In Lisp parlance, we replaced setq's by setf's. Extra analysis machinery and tables of setters and getters (updates and accessors) were needed.

4.6 Statistics

The table in `Figure 13` records some of the progress made in 2012. We selected representative Mark&Sweep codes generated at various time points during the year. Each collector was run against a random-based mutator whose input includes an upper bounds on the maximum number of nodes in the heap (see row 2 in the table).

The runtime column gives total runtime before the system ran out of memory, including allocation, pointer-swinging, and collection times. The time spent in garbage collection is typically on the order of 10-20%.

	runtime	gc time	runtime	gc time	runtime	gc time	alloc n/s	runtime	gc time	alloc n/s
nodes:	1000		10000		100000			1000000		
2012										
12-Mar	>600									
25-Apr	138	22								
17-May	0.031	0.005	1.6	0.5	38	10.6	29k			
18-Jun			0.73	0.007	1.38	0.16	133k			
17-Aug			0.04	0.017	1.2	0.22	162k	47.5	2.75	42k
14-Sep			0.09	0.017	1.2	0.28	168k	61	6.6	33k

9-Oct					3.9	0.58	292k	99	16	114k
9-Nov							410k			200k

Figure 13: Runtime Measurements

The runtime and GC times however are not a reliable indicator because the mutator itself changed during the year. The columns labeled Allocation n/s (nodes/second) provide better indicators of progress. They show the rate at which nodes are allocated and collected, showing steady progress. By the end of the project, the collection performance was up to 300k nodes/second for a 1M node memory. There are still many optimizations that can be applied to further improve performance.

4.7 Proof generation results

We extended the following transformations to emit proofs at application time: Simplification, Observer Maintenance, Observer Implementation, StructureEx, finalizeCotype, and others.

Transformations	lines of ISAR/Isabelle proof script
Algorithm Design: fixpoint iteration	300 + 10(library)
Simplification	45
Observer Maintenance: WS	3280
Observer Refinement of payload	1748
Observer Refinement: tgt	1706
Observer Refinement: outNodes	680
Observer Refinement: roots	2275
Observer Maintenance: rootCount	1791
Observer Refinement: nodes	2110
Import random mutator	14348
Simplify	107
Observer Refinement: supply	2010
Observer Maintenance: supplyCount	2291

Figure 14: Proof-Generation Results (part 1)

We made further progress on generating proofs automatically from the various transformation steps in our Mark&Sweep derivation. We were able to generate proofs from most of the transformations used in our Mark&Sweep derivation, as shown in Figure 14 and Figure 15. The left column lists the transformations of the derivation and the right lines lists the number of lines of Isabelle/ISAR proof script emitted by the transformation, providing a proof of the correctness of the generated refinement. Most of the proofs are instance-level proofs, but some are a mix of library proofs (such as the proof of the algorithm design theory) and instance-level details. The blue counts are a sum of the counts above, since the corresponding transformation has to recapitulate previous steps for technical reasons (which we hope to obviate). Overall, the transformations automatically generate over 33,000 lines of machine-checkable proof script. This is a major result of the project. We have demonstrated that a derivational approach to algorithm generation can produce proofs as a by-product and that *the marginal cost of producing those proofs is effectively zero*.

Transformations	lines of ISAR/Isabelle proof script
Observer Refinement: black->blackCM	4200
Observer Refinement: WS->WL	3846
Observer Refinement: WL->WStack	4242
FinalizeCoType Memory	33056
Definitions	
Type Isomorphism: Memory	
Type Isomorphism: Optimizations	
DataType Refinement: Maps->Vectors	library
DataType Refinement: Stacks->Vectors	library
DataType Refinement: Sets->Lists	library
Optimizations	
Threads and Mutexes	library
Globalize Memory	
Code Generation	not yet

Figure 15: Proof Generation Results (part 2)

4.8 Flex Seedling

The graph in Figure 16 shows the results of running successive executable versions of Flex over a representative set of our tests. As more optimizations are applied to Flex, the number of theorems proved by Flex increases, and the number of theorems not proved by Flex (due to reaching a timeout) decreases.

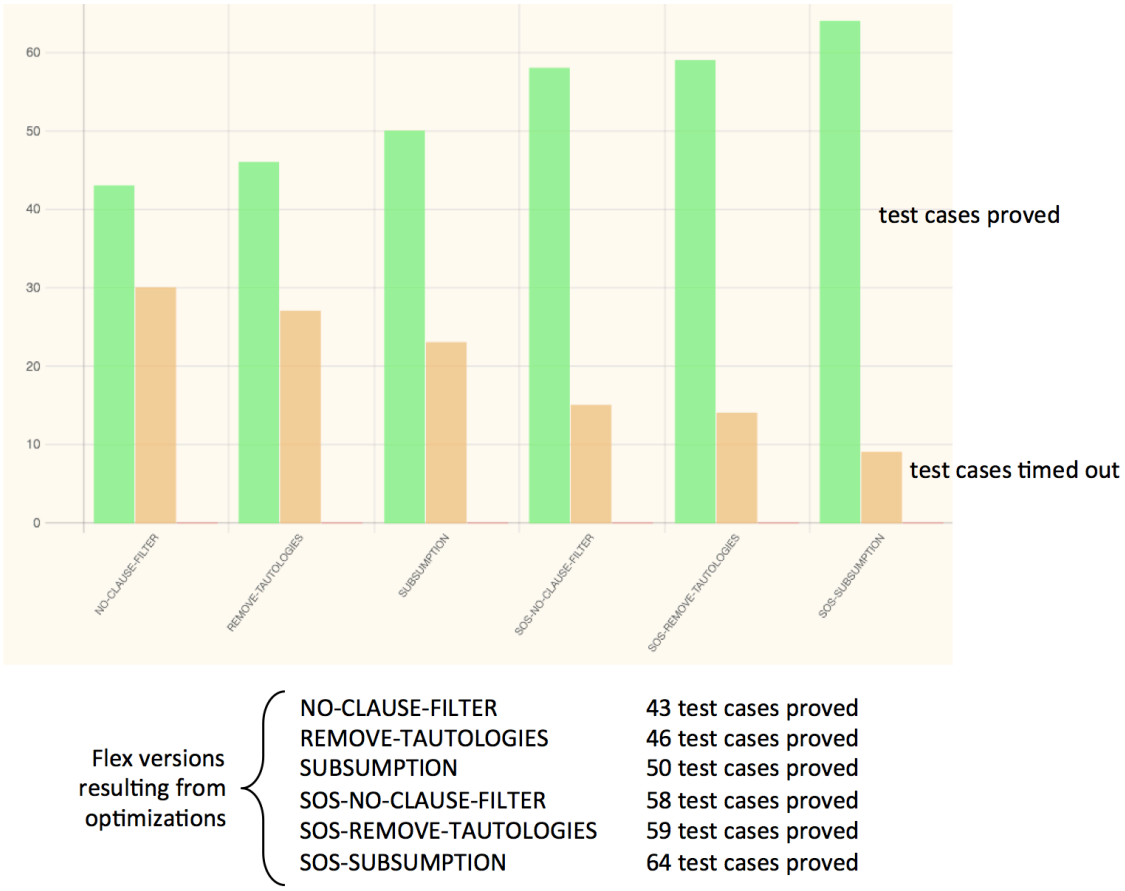
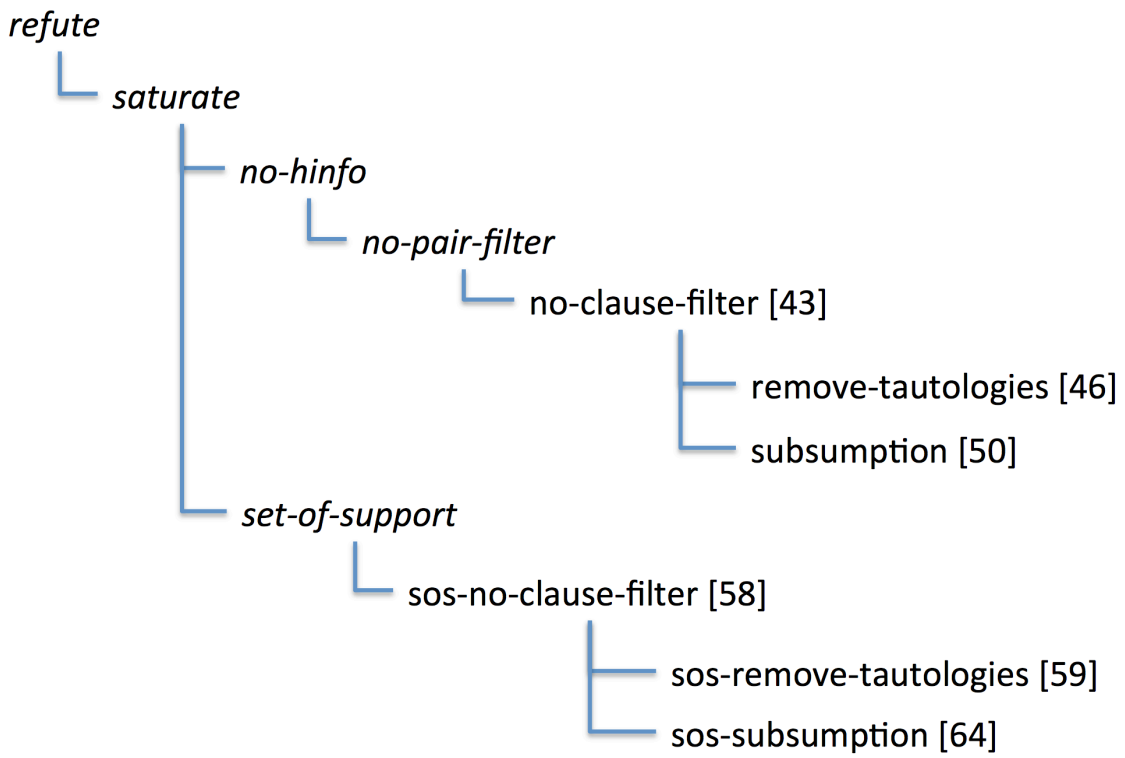


Figure 16: Flex test cases

The diagram in shows the derivation tree for Flex, rooted at the top-level declarative specification.



The numbers in […] are the number of tests that pass (i.e. number of successful proofs).
The versions in *italics* are not executable and so they have no numbers next to them.

Figure 17: Flex Derivation tree

5 CONCLUSIONS

5.1 Synthesizing Concurrent Garbage Collectors

We conclude with some reflections on the synthesis of concurrent garbage collectors. Perhaps the main contribution of this work is a robust methodology and tool support for the automated generation of a family tree of programs that are correct-by-construction and have proofs automatically generated as a by-product of the refinement process. A remarkable aspect of our derivation trees is that they are mainly built from highly reusable design theories (e.g. the fixpoint iteration theory applies to a wide range of problems), transformations, and library refinements. Not only are the design theories and transformations applicable across domains, but they provide a compelling explanation of seemingly complex algorithms – Dijkstra's on-the-fly concurrent Mark&Sweep algorithm was discovered after many flawed attempts [Dijkstra78], but it falls our naturally via fixpoint iteration and Observer Refinement. Moreover, it has been our intention in designing these specifications and transformations that they require relatively simple calculation sequences to reach useful results. We hope that the reader is convinced of this by the examples in this report, which are representative.

The goal of the CRASH program has been to develop clean-slate approaches to enhancing security in a computer host. We have contributed by developing techniques for recording our calculations and using them to generate checkable proofs of the correctness of our derivations. Absence of many of the vulnerabilities that are rife in conventionally produced software (e.g. buffer overflows and null pointer dereferences) is a checkable feature of our approach. By automating the production of proofs, we lower the cost of providing high levels of assurance as a normal part of software development. We demonstrated that a derivational approach to algorithm generation can produce proofs as a by-product and that *the marginal cost of producing those proofs is effectively zero*.

A crucial aspect of software engineering that is rarely addressed in formal approaches is the cost of code maintenance and evolution. In our formal approach, there should be no maintenance in the sense of bug fixing, but there will always be a need to adapt to changing requirements and changing design decisions. Somewhat to our surprise, we found that having an explicit executable metaprogram to generate code + proof, gave us an opportunity to explore code evolution at the proper level – in terms of modifying requirements and modifying design decisions in the metaprogram itself, rather than in

informal design discussions and documents. We found that a derivation for one GC algorithm provided most of the design DNA needed for other GC algorithms. In this case we were not modifying the problem requirements (which are common and fixed), but essentially making alternative design choices. In this case the metaprogram modifications were done manually, but the way is open to more automated approaches.

5.2 Flex Seedling

We believe that the results of this Flex seedling show promise towards the goal of building a self-adaptive theorem prover.

6 References

[AAR15] Association for Automated Reasoning, Newsletter No. 111, June 2015. *http://www.aarinc.org/Newsletters/111-2015-06.html*

[ACL2] The ACL2 Theorem Prover. *http://www.cs.utexas.edu/users/moore/acl2*

[Blaine94] Blaine, L. and Goldberg, A., DTRE: A Semi-Automatic Transformation System, in *Constructing Programs from Specifications*, Ed. B. Moeller, North-Holland, 1991, 165—204.

[CaiPaige89] Cai, J., and Paige, R. Program Derivation by Fixed Point Computation, Science of Computer Programming 11, 3 (April 1989), 197—261.

[Dijkstra78] Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., and Steffens, E., On-the-fly garbage collection: An exercise in cooperation. Communications of the ACM 21, 11 (November 1978), 965—975.

[Green69] Cordell Green “The Application of Theorem Proving to Question-Answering Systems.” Ph.D. thesis, Stanford University, 1969.

[He86] He, Hoare, and Sanders, Data Refinement, Refined, 1986

[Liu13] Liu, Y. *Systematic Program Design: From Clarity to Efficiency*, Cambridge Univ. Press, 2013

[Jacobs97] Jacobs, B., and Rutten, J., A tutorial on (co)algebras and (co)induction. Bulletin of the European Association for Theoretical Computer Science 62 (1997), 222—259.

[Kleene56] Kleene, S, *Introduction to Metamathematics*, American Mathematical Society Press, 1956.

[Kreitz12] Locales for Fixpoint Iteration Algorithm Theory, Kestrel Institute Technical Report, 2012.

[Mossakowski06] Mossakowski, T., Reichel, H., Roggenbach, M., and Schroeder, L. Algebraic-coalgebraic specification in CoCasl, J. Logic Algebraic Programming 67 (2006).

[Paige82] Paige, R. and Koenig, S., Finite Differencing of Computable Expressions, TOPLAS 4(3), 1982, 402-454.

[Pavlovic10] Pavlovic, D., Pepper, P., and Smith, D.R. Formal derivation of concurrent garbage collectors. In *Proceedings of 10th International Conference on Mathematics of Program Construction (MPC 2010),* Springer Verlag LNCS 6120, pp.353—376.

[Pelletier86] Francis Pelletier, "Seventy-Five Problems for Testing Automated Theorem Provers", *Journal of Automated Reasoning*, 2(2):191-216, 1986.

[Robinson65] Alan J. Robinson, “A Machine-Oriented Logic Based on the Resolution Principle.” *Journal of the ACM* 12(1):23–41, 1965.

[Rothe01] Rothe, J., Tews, H., and Jacobs, B. The coalgebraic class specification language CCSL, Journal of Universal Computer Science 7, 2 (2001), 175—193.

[Rutten00] Rutten, J. Universal coalgebra: a theory of systems, Theoretical Computer Science 249, 1 (2000), 3 — 80.

[SmithD15PET] Kimmell, G., Smith, D., Westbrook, E., and Westfold, S. Proof-emitting transformations, Tech. Rep. KES.U.15.2, Kestrel Institute, 2015.

[SmithD03] Pavlovic, D., Pepper, P., and Smith, D.R. Colimits for concurrent collectors. In *Verification: Theory and Practice: Festschrift for Zohar Manna (2003),* N.Dershowitz, Ed., LNCS 2772, pp.568—597.

[SmithD9009] Smith, D.R. KIDS — a semi-automatic program development system, IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16, 9 (1990), 1024—1043.

[Specware03] Kestrel Institute, Specware System and documentation, 2003. http://www.specware.org/.

[SmithD9305] Smith, D.R. Constructing specification morphisms, Journal of Symbolic Computation, Special Issue on Automatic Programming 15, 5-6 (May-June 1993), 571—606.

[SmithD9302] Smith, D.R. Pushouts preserve conservative extensions: Another proof of the modularization theorem, Tech. Rep. KES.U.93.1, Kestrel Institute, February 1993.

[SW03] Specware System and documentation, 2003, Kestrel Institute, http://www.specware.org/

[Tarski:1955] Tarski, A. A lattice-theoretical fixpoint theorem and its applications, Pacific J. Math. 5, 2 (1955), 285—309.

[TPTP] The "Thousands of Problems for Theorem Provers" Library. *http://www.tptp.org*

[Turski87] Turski, W.M., and Maibaum, T.E. *The Specification of Computer Programs,* Addison-Wesley, Wokingham, England, 1987.

[Veloso95] Veloso, P.A., and Maibaum, T. On the modularization theorem for logical specification, Information Processing Letters 53, 5 (1995), 287—–293.

[Whalen02] Whalen, M., Schumann, J., and Fischer, B. Synthesizing certified code, in Proc. Formal Methods Europe (FME 2002) (Copenhagen, Denmark, 2002), Springer LNCS 2391, pp.431—450.

[Wos00] Larry wos and Gail W. Pieper, "A Fascinating Country in the World of Computing – Your Guide to Automated Reasoning", World Scientific Publishing Company, 2000.

7 LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

cpo	complete partial order
CRASH	Clean-Slate Design of a Secure Host
DARPA	Defense Advanced Research Projects Agency
GC	Garbage Collector
HACMS	High-Assurance Military Systems
VDM	Vienna Development Methodology