Formal Coalgebraic Specifications and their Refinement

Douglas R Smith
Stephen Westfold

Kestrel Institute
Palo Alto, California

www.kestrel.edu
Specware: Software Development by Refinement

semantics: $\mathsf{SPEC}^{op}$ \rightarrow \mathsf{CAT}$

refinement in $\mathsf{SPEC}$

code generation

$\text{Spec}_0 \rightarrow \text{denotes} \rightarrow \text{Spec}_1 \rightarrow \text{Spec}_2 \rightarrow \ldots \rightarrow \text{Spec}_n \rightarrow \text{Code}$

category of models for $\text{Spec}_0$

models for $\text{Spec}_1$

models for $\text{Spec}_2$

models for $\text{Spec}_n$

a model for $\text{Spec}_n$
Refinement Sequence for Garbage Collection

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

Model Construction Tool

Library of resource and task models

Model of Scheduling Problem

Scheduler Generator

Customized Scheduler (in MetaSlang)

Optimization and Code Generation

Customized Scheduler (in CommonLisp)

250 lines

6560 LOC
632 defs

19088 LOC
1784 defs

input data

Customized Scheduler

C0    C2    C4    C6     C8
schedule
Deriving Common Garbage Collection Algorithms

Collector Spec

- Reference Count Collectors
  - maintain count of predecessors

- Tracing Collectors
  - recalculate current set of dead nodes
    - partitioned memory model
    - monolithic memory model
      - Copying Collectors
      - Marking Collectors
        - Generational Collectors
        - Mark & Compact ± generations
        - Mark & Sweep ± generations

Reference Count Collectors

Generational Collectors
Model of Memory as a Rooted Graph

preroots

registers

globals, static

stack

heap

...
State Machine Models: Mutator + Collector

Mutator is an application that allocates heap nodes, and manipulates arcs (pointers).

Collector identifies dead nodes and recycles them.

A node is dead if there are no paths to it from the roots

$$n \in \text{dead} \iff \text{paths(roots, n)} = \{\}$$

Requirements

Safety: No active nodes are ever collected

Transparency: Throughput, pause times, footprint, promptness
Algebra and Coalgebra

an algebra is a morphism $F A \rightarrow A$
where $F$ is a (commonly polynomial) functor
$F$ provides the signature of operations of the algebra

a coalgebra is a morphism $A \rightarrow F A$
where $F$ is a (commonly linear) functor

coalgebras provides a unifying treatment of
- dynamical systems
- automata
- transition systems
Specifying Algebraic Types

An algebraic type is defined by constructors
  • well-founded
  • new functions defined inductively over constructors

type List a = nil | cons a (List a)

op length: List a → Nat
  length nil = 0
  length (cons a lst) = 1 + length lst

length is defined in terms of its value over the constructors

List is defined using constructors nil and cons
Specifying Coalgebraic Types (aka cotypes)

A coalgebraic type is characterized by observers

- not well-founded: may be circular or infinite
- transformers specified coinductively by effect on observers

type Graph
op nodes: Graph → Set Node
op outArcs : Graph → Node → Set Arc

op addArc(G:Graph) (x:Node, y:Node | x,y ∈ nodes G) :
    {G’:Graph | nodes G’ = nodes G
    & outArcs G’ x = (outArcs G x) + (x→y) }

addArc is specified in terms of its effect on the observers

Graph is specified using observers nodes and outArcs
Coalgebraic Specifications

- Algebraic types used for ordinary data (boolean, Nat, List)
- Coalgebraic types used for state (heaps, objects), streams
- Observers \( \text{obs} : \text{State} \rightarrow A \)
  - basic/undefined
  - defined but maintained
  - defined but computed
- Transformers \( t : \text{State} \rightarrow \text{State} \)
  - preconditions
  - postconditions: coinductive constraints on observations
Coalgebraic Refinement

- Coalgebraic types remain undefined until the last step
- Observer transformation:
  - Introduction
  - Maintenance (of a definition)
  - Refinement (to a more concrete observer)
- Transformer refinement
  - Postconditions are strengthened
  - Synthesized to a definition at the last step
Tracing Collectors:
Instantiated Small-Step Fixpoint Iteration

\[ S \leftarrow \{ \} \]
\[ \text{while } \exists z \in (\text{roots}(G) \cup \text{sucs}(G)(S)) \setminus S \text{ do} \]
\[ S \leftarrow S \cup \{z\} \]
return S

to optimize the algorithm, we introduce a new observer:

\[ WS \; G = (\text{roots} \; G \cup \text{sucs}(G)(S)) \setminus S \]
Maintaining Observers

Observer Maintenance Transform (aka Incrementalization, Finite Differencing)

• given a defined observer

\[ WS (G:Graph):\text{Set } A = e \ G \]

• for each transformer \( t \), add definition to postcondition:

\[ t(G:Graph \mid WS \ G = e \ G ):\]

\[ \{ G':Graph \mid \ldots \land WS \ G' = e \ G' \} \]

• simplify
Maintaining Observers

type Graph
op nodes: Graph \rightarrow \text{Set Node}
op outArcs : Graph \rightarrow \text{Node} \rightarrow \text{Set Node}
op roots : Graph \rightarrow \text{Set Node}
op S : Graph \rightarrow \text{Set Node}

op \text{WS}(G:\text{Graph}):\text{Set Node} = (\text{roots } G \cup \text{outArcs } G (S G)) \setminus (S G)

op addArc(G:Graph) (x:Node, y:Node) :
\{G':Graph | \text{nodes } G' = \text{nodes } G
\land \text{outArcs } G' x = (\text{outArcs } G x) + (x \rightarrow y)
\land \text{WS } G' = \text{WS } G \cup \{y | x \in S G \land y \notin S G\}\}

design-time calculation:

\text{WS } G' = (\text{roots } G' \cup \text{outArcs } G' (S G)) \setminus (S G)
= (\text{roots } G \cup \text{outArcs } (G \cup \{x \rightarrow y\}) (S G)) \setminus (S G)
= (\text{roots } G \cup \text{outArcs } G S) \setminus (S G) \cup \{y | x \in (S G)\} \setminus (S G)
= \text{WS } G \cup \{y | x \in (S G) \land y \notin (S G)\}
after all design-time calculations to enforce the invariant:

\[
\text{invariant } WS = (\text{roots } \cup \text{outArcs}(S)) \setminus S
\]

\[
\text{atomic } \langle S \leftarrow \{\} || WS \leftarrow \text{roots} \rangle
\]

\[
\text{while } \exists z \in WS \text{ do}
\]

\[
\text{atomic } \langle S \leftarrow S \cup \{z\} || WS \leftarrow WS \cup \text{outArcs}(z) \setminus S - z \rangle
\]

\[
\text{return } S
\]

\[
\text{atomic } \langle \text{addArc}(x,y) || WS \leftarrow WS \cup \{y | x \in S \land y \notin S} \rangle
\]

dthis is essence of the coarse-grain Dijkstra et al. “on-the-fly” collector
Generating Proof Scripts

For example, a refinement based on this calculation from the derivation of a Mark & Sweep garbage collector:

<table>
<thead>
<tr>
<th>Sequence of Rewrites</th>
<th>Justification for Each Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialState x0</td>
<td>unfolding initialState</td>
</tr>
<tr>
<td>= FHeap x0 {}</td>
<td>unfolding FHeap</td>
</tr>
<tr>
<td>= roots x0 ∪ allOutNodes x0 {}</td>
<td>rule allOutNodes_of_emptyset</td>
</tr>
<tr>
<td>= roots x0 ∪ {}</td>
<td>rule right_unit_of_union</td>
</tr>
<tr>
<td>= roots x0</td>
<td></td>
</tr>
</tbody>
</table>

would also automatically generate this Isabelle/Isar proof script:

```isar
definition initialState_refine_def where "(initialState x0) = (roots x0)"
proof -
  have " (initialState x0) = FHeap x0 {}" by (unfold initialState_def, rule HOL.refl)
  also have "... = (roots x0 ∪ allOutNodes x0 {})" by (unfold FHeap_def, rule HOL.refl)
  also have "... = (roots x0 ∪ {})" by (rule_tac f="?term ∪ y" in arg_cong, rule allOutNodes_of_empty_set)
  also have "... = (roots x0)" by (rule union_right_unit)
  finally show ?thesis .
qed
```

The proof script discharges the proof obligation of the refinement


Coalgebraic Specification and Refinement