STARFISH: A TABLE-CENTRIC TOOL FOR DESIGN DERIVATION

by

Alexander W. Tsow

Submitted to the faculty of the University Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the Department of Computer Science

Indiana University

July 2007

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Steven D. Johnson
(Principal Advisor)

R. Kent Dybvig

Doctoral

Committee

Lawrence S. Moss

May 22, 2007

Amr Sabry

Acknowledgements

This thesis would not have been possible without the patience, insight and encouragement of my supervisors, my colleagues, and my family. I would like to thank Steven D. Johnson for his unflagging commitment to guiding and supporting me through my graduate study. He has shared both his love of computer science and personal wisdom with me. I am grateful for it. I thank R. Kent Dybvig, Lawrence S. Moss, and Amr Sabry for their inspirational teaching, perspectives, and commentary on the drafts of this work.

NASA's *Graduate Student Research Program* fellowship supported the bulk of Starfish's software development. Thanks to Paul Miner, Ricky Butler, and the NASA Langley Research Center for sustaining this research. I also want to thank Indiana University, the NSF, Google and the I3P for their role in underwriting my graduate education. I am further indebted to L. Jean Camp for pushing me through the finish line of this very difficult process.

My parents, Dorothy and Bill, have given me the encouragement, means, and opportunity to live a fulfilling life in all respects. My cherished wife, Amy Goldenberg, deserves far more than a simple acknowledgment for her role. I thank her

here; however the best way to express my gratitude is by reciprocating her unerring devotion.

Abstract

Behavior tables are a visual formalism for representing synchronous systems of communicating processes. Although behavior tables arose from hardware modeling methods, they operate on arbitrarily abstract data-types. Originally conceived as an aid for imposing architecture on behaviorally oriented specifications, behavior tables inherited a structural algebra from the *Digital Design Derivation (DDD)* system. This thesis extends the algebra in three ways. It incorporates a transformation for retiming operations. It adds serialization by extending the notion of correctness to include stuttering alignments. It introduces mechanisms for declaring and refining abstract data types. This thesis further contributes *serialization tables*—a complementary behavior table form—for assisting interactive construction of *schedules*. A prototype tool, *Starfish*, implements these technologies. Two medium-scale examples—an SECD machine derivation and an abstract signal factorization for a hardware garbage collector—demonstrate feasibility of these techniques in non-trivial systems.

Contents

Chapter 1

Introduction

Engineering is an interactive process that requires intelligent interaction at many levels. This thesis advances an engineering discipline for high-level synthesis and architectural decomposition that integrates perspicuous representation, designer interaction, and mathematical rigor. *Starfish*, the software prototype for the design method, implements a table-centric transformation system for reorganizing control-dominated system expressions into high-level architectures. Based on the *digital design derivation (DDD)* system [60]—a designer-guided synthesis technique that applies correctness preserving transformations to synchronous data flow specifications expressed as co-recursive stream equations—Starfish enhances user interaction and extends the reachable design space by incorporating four innovations: behavior tables, serialization tables, data refinement, and operator retiming.

1.1 Behavior Tables

Behavior tables express systems of co-recursive stream equations as a table of guarded signal updates. Developers and users of the DDD system used manually constructed

behavior tables to help them decide which transformations to apply and how to specify them. These design exercises produced several formally constructed hardware implementations: the FM9001 microprocessor [12, 11], an SECD machine for evaluating LISP [103], and the *SchemEngine*, garbage collected machine for interpreting a byte-code representation of compiled Scheme programs [16, 56]. Bose and Tuna, two of DDD's developers, have subsequently commercialized the design derivation methodology at *Derivation Systems, Inc. (DSI)*. DSI has formally derived and validated PCI bus interfaces [13], a Java byte-code processor [9], and prototypes of SPIDER [30]—NASA's ultra-reliable communications bus.

To date, most derivations from DDD and DRS have targeted hardware due to its synchronous design paradigm. However, Starfish expressions are independent of the synchronization mechanism; there is no commitment to hardware or globally broadcast clocks. Though software back-ends for design derivation are limited to the DDD stream-interpreter, targeting synchronous or real-time software is not substantively different from targeting hardware.

The separation of concerns [95, 57]—e.g., architecture, behavior, data representation, and interface coordination—is standard engineering doctrine. In particular, it is futile to expose all aspects equally well with a single language. Behavior tables represent a compromise between behavior and architecture: its rows roughly characterize a specification's control oriented aspects, while the columns represent its architectural, or structural, aspects. Figure 1 shows two different, but equivalent behavior tables. A degenerate table simply reformats stream equations to appear vertically, rather than horizontally. The other table expands selector terms into the decision

Inputs: (go, a, b)
Outputs: (done, acc)

state:Seq.	u:Seq.	v:Seq.	acc:Seq.	done:Comb.
(sel state (sel go zu idle) (sel (zero? u) idle zv) (sel (zero? v) idle shift) zv)	(sel state (sel go a #) u u (* u 2))	(sel state (sel go b #) v v (/ v 2))	(sel state (sel go 0 acc) acc acc (sel (even? v) acc (+ acc v)))	(sel state (sel go false true) false false false)

Inputs: (go, a, b)
Outputs: (done, acc)

go	state	(zero? u)	(zero? v)	(even? v)	state:Seq.	u:Seq.	v:Seq.	acc:Seq.	done:Comb.
false	idle	#	#	#	idle	#	#	acc	true
true	idle	#	#	#	zu	a	b	0	false
#	zu	true	#	#	idle	u	v	acc	false
#	zu	false	#	#	zv	u	v	acc	false
#	zv	#	true	#	idle	u	v	acc	false
#	zv	#	false	#	shift	u	v	acc	false
#	shift	#	#	true	zv	(* u 2)	(/ v 2)	acc	false
#	shift	#	#	false	zv	(* u 2)	(/ v 2)	(+ acc v)	false

Figure 1: Behavior tables for equivalent systems. The upper table is a degenerate form, approximating stream equation expressions. The lower table enhances control expression with the decision table which guards sets of simultaneous actions.

table, clarifying the control flow and aligning simultaneous actions. The behavior table transformations—among other things—allow designers to trade between these two axes, thereby balancing between the two aspects. It is no surprise, then, that behavior tables are well suited for deriving architectural components from behaviorally oriented expressions.

1.2 Type Management and Data Refinement

Behavior tables operate on arbitrarily abstract data-types, not just bit-vectors and bounded integers. In this respect, they are far more expressive than standard hardware description languages. Starfish implements an explicit type system and a framework for data refinement to support high-level specification with abstract data types.

Demand for explicit typing arose from several areas: the need to limit decision expressions to finitely branching guards, the need to prevent incompatible signal merging opportunities among unused slots in table columns, and the desire to increase feedback by disallowing unsound transformations at earlier stages. The type system, which is based on multisorted structures, takes on a second responsibility: it forms a database of term-level identities. One of the core transformations, the *replacement rule* (p. 51) applies algebraic identities (e.g., operator commutativity) to terms. While many term rewrites in DDD are combinator expansions, each algebraic term rewrite requires external validation. Starfish leverages the type system's identity database to confirm algebraic rewrites—only the identity pattern needs external verification.

Since the type system declares function symbols, signatures and identities, it provides a foundation for *data refinement*. At the simplest level, a system of identities can express one-to-one homomorphisms between types. While such an identity system transforms abstract *terms* into representation terms, the architectural algebra preceding Starfish [61] could not transform abstract *signals* into representation signals in a general way. While the first attempts to impose signal-level refinement were *ad hoc*, Starfish's refinement process follows from the retiming transformation (p. 53)

and recursive identity expansion. In addition to refinement by one-to-one homomorphism, this thesis presents methods deriving two other implementation patterns: *one-to-many refinements*, where there are multiple representations for each abstract type, and *stateful refinements*, which represent multiple signals with references to a shared store.

While behavior tables are not useful for defining data refinements, they are useful for exploiting or managing their consequences. Data refinements lead to more detailed specifications and consequently a wider transformation space. System decompositions, the problem for which behavior tables were invented, may "cut across" a representation that implements an abstract type with a collection of signals. For instance, suppose a refinement simulates abstract stacks with a pointer and array; subsequent architectural organization may separate the array from the pointer. In another case, a stateful refinement may impose serial access on the previously unconstrained concurrency of abstract operations. Behavior tables and their scheduling aid, serialization tables, provide an interactive method for integrating the serial requirements into a system's control and architecture by scheduling access before and after stateful refinements—this is the principal challenge of the SECD derivation in Chapter 9.2.

1.3 Scheduling and serialization

Starfish introduces *serialization tables* for scheduling the evaluation of complex action terms over several steps. Like behavior tables, columns represent signals and rows

represent simultaneous actions which update the signals. Serialization tables are an organizational aid that helps designers solve the NP-hard problems involved in high-level synthesis [70, 19]; e.g., how to fit an evaluation sequence within a specified number of registers and execution units. Serialization tables help specify evaluation order and intermediate resource usage for compound actions in a behavior table. As the schedule develops, the serialization tables display partial symbolic-evaluations of the intermediate actions. This feedback mechanism helps designers specify actions in the subsequent steps. Starfish validates correctness before integrating the actions into behavior table expressions.

The DDD algebra views serialization as primarily a behavioral problem. Yet, the goal of scheduling is often architecturally dictated. One must use a limited set of resources. Register allocation, functional allocation, and timing are not fully exposed in DDD's behavioral representations. Serialization tables display these aspects more clearly than DDD's co-recursive stream equations, making them a better suited medium for the schedule specification process.

In addition to specifying schedules for behavior table actions, serialization tables may re-visit a sub-schedule of a previously serialized action. This is often necessary after data-refinements expose more detail. For instance, a push onto a stack may require incrementing the stack pointer and allocating extra memory. These operations must conform to the architectural constraints—perhaps the memory or incrementing functional unit are in use during the step where data-refinement has inserted these actions. Re-serializing the local sub-schedule focuses attention on this task without obfuscating it with other parts of the pre-existing schedule.

1.4 Retiming

Starfish supports retiming in two ways. One is with serialization tables and local re-serialization, as introduced above. The other is with a transformation that converts combinational signals to sequential signals and vice versa. In schematic terms, the transformation pushes a functional unit from one side of a register to the other. Although retiming is the critical step in transforming abstract signals to representation signals, the motivating example in Starfish was the stack-calculator introduced in Example 3.17 (p. 46). The specification used a combinational `top` accessor for the output signal. Any pseudo-realistic implementation would store the top value in a register. The exercise of hand-specifying a stack-calculator with a registered `top` signal was enough to see a generalizable pattern. Indeed, equivalent transformations have been used in formal synthesis [23] and microarchitecture algebra [66].

1.5 Organization

The remainder of this thesis is organized as follows: Chapter 2 examines related research. Chapter 3 presents the basic stream modeling formalism and stream transformations that underlie behavior tables. The type system, based on multisorted structures, is presented here. After setting context with DDD's architectural algebra, the chapter shows how the retiming transformation generalizes the combinational identification rule to sequential signals. Chapter 4 introduces behavior table notation

and its transformation algebra. Chapter 5 gives an overview of the Starfish implementation, including the heavily used macros, the software organization, the component addressing system, the table transformation commands, the display mechanism, and the interprocess communication. Chapter 6 shows how to perform system factorization with behavior tables. Higher level commands with exploration heuristics facilitate this process. Chapter 7 defines serialization and the tabular worksheet that helps the designer specify correct schedules of operations. Chapter 8 presents the work on data refinement. It first presents the justification for the one-to-one case, then generalizes to the one-to-many and stateful cases. The chapter ends with an example that combines all previous techniques to transform a simple stack calculator specification into an architecture with explicit controller, ALU, and memory. Chapter 9 demonstrates Starfish's feasibility on larger designs with two case studies. In the first, a stop-and-copy garbage collector specification using abstract memory signals is factored into a controller and dual-ported memory process. A data refinement replaces literal value "swapping" between abstract memory signals—an "old" and a "new"—with a switch indicating which signal represents "old" and "new." A second case study derives a low-level specification for an SECD machine from a high-level specification of LISP operational semantics. This case study complements the work of Robert M. Wehrmeister [103], who used DDD to derive hardware from a low-level architecture similar to the result of this case study. Chapter 10 concludes the thesis with lessons learned and suggestions for future work.

Chapter 2

Related Research

2.1 Design Derivation

Johnson applies a functional modeling formalism to digital system synthesis [52, 51, 53]. His approach was motivated by early work of Friedman, Wand and Wise on applicative programming for systems which developed a programming style where computational processes are represented as recursive stream networks [28, 29, 50].

Johnson adopted the same programming dialect to model digital networks [26, ch. 12], and extended standard program transformation strategies to target digital implementations [102, 99, 51]. He extended Wand's approach to compiler construction [100, 101] which involved isolating lambda-combinators to identify primitive operations, and factoring the language semantics into a recursive compiler and an iterative "machine" component expressed in terms of the primitives.

Specifying instruction set processors (ISPs) as first-order recursion equations, Johnson developed formal constructions that translate the machine model to an abstract, synchronous-sequential architecture, which is further transformed in a functionally equivalent, realistic digital system description. Bose, Boyer, Rath, and others implemented the formalism as the *DDD transformation system* [58, 62, 59]. DDD was

used to do a series of demonstration designs, including a garbage collector [58, 14], an SECD computer [103], a byte-code interpreter for compiled Scheme [16], a derived implementation of Hunt's formally verified FM-9001 CPU [12, 11], a fault tolerant clock sychronization circuit [72, 73].

DDD transformations operate on two different system representations: 1) a *functional (or behavioral) specification*—modeled as a functional subset of the Scheme programming language—that expresses behavior in a time-oblivious manner and 2) an *architectural specification* that expresses linear tail-recursive functions as a synchronous stream systems where evaluating each recursive call corresponds to a step in the stream network. DDD's algebra over behavioral specifications preserves the time-oblivious semantics of Scheme, while its algebra over architectural specifications preserves observable stream outputs—hence is time-sensitive. This form of architectural equivalence is stricter than behavioral equivalence. In particular, the behavioral algebra includes serialization transformations which stretch the resulting system's output timeline; the system takes more steps to produce its result.

Behavior tables (Starfish's system representation) directly encode DDD's architectural language. Starfish's core transformations begin with transliterations of DDD's architectural algebra. Starfish adds a retiming transformation and serialization. Retiming preserves the observational equivalence, however serialization breaks this property by stretching the output timeline. Starfish's serialization is consistent with DDD's behavioral equivalence. This thesis submits that the motivation for serialization is to optimize architectural aspects (such as the number of execution units in a circuit), and thus behavior tables—which exposes architecture—are a more apt

representation to serialize than DDD's behavioral expressions.

2.2 Type Management

In DDD, function and constant symbols are declared in the top level scope and given a specific interpretation using a functional subset of Scheme. This enables DDD to simulate system descriptions by stepwise execution of the stream expressions, using Scheme's `eval` to interpret terms. A representation file consists of a flat list which pairs symbol names with their Scheme interpretation. DDD evaluates type consistency using an *abstract interpretation* [22] which replaces constant symbol interpretations with the appropriate type token and function symbols with an expression that checks the intended signature—a similar approach to that taken by O'Donnell's Hydra [75].

For example, the following file defines the terms and functions for a DDD blackjack dealer case study [59].

```
(define tt         (bit "1"))
(define ff         (bit "0"))
(define ?          (bvec "00000"))
(define zero       (bvec "00000"))
(define one        (bvec "00001"))
(define sixteen    (bvec "10000"))
(define twentyone  (bvec "10101"))
(define 10pace     (bvec "01010"))
(define -10pace    (bvec "10110"))

(define get        (bvec "00"))
(define add        (bvec "10"))
(define use        (bvec "01"))
(define tst        (bvec "11"))
```

```
(define cd-?      (bvec "00000"))
(define out-?     (bvec "00000"))

(define abs-add_inst-nop   (bit "0"))
(define abs-add_inst-addto (bit "1"))
(define abs-add_port_a-?   (bvec "00000"))
(define abs-add_port_b-?   (bvec "00000"))

(define addto  (lambda (x y) (nat-to-v (+ (v-to-nat x) (v-to-nat y)) 5)))

(define or?    (lambda (x y) (b-or x y)))
(define >?     (lambda (x y) (v-gtp x y)))
(define ace?   (lambda (x)   (v-eqp x one)))
```

Evaluating type consistency would use a representation file that redefines the constants `?`, `zero`, `one`, etc. as the symbol `bvec`, and the function `>?` as the error-throwing lambda expression:

```
(define >?
 (lambda (x y)
  (if (and (eq? x 'bvec) (eq? y 'bvec))
    'bit
    (error '>? "Type Mismatch: >?(~s,~s)" x y))))
```

Zhu and Johnson proposed leveraging multisorted structures as a basis for term manipulation in DDD [107]. Their stack calculator example, which they develop over abstract and implementation types, was a catalyst for the data-refinement mechanisms that Starfish implements. Starfish declares function signatures and constant types (precise details presented in Section 3.2) with multisorted structures [68], however it does not attach an interpretation to the symbols as DDD does. Since terms are first-order, this static information suffices for type consistency checking. While automated type consistency checking is one benefit to Starfish's type declaration regimen,

its primary purpose is to facilitate term-level algebraic reasoning via the replacement rule in behavior tables (p. 68) and the `ser-eval-ident` substitution in serialization tables (p. 142).

In addition to declaring symbol signatures, Starfish's type declarations also include identities. These identities constrain the space of interpretations; since Starfish does not insist on a specific one, the terms are said to have *loose semantics* [34, 3]. Sort identities are universally quantified over typed formal variables, and Starfish will apply the identity to arbitrary terms that match either the left-hand-side pattern or the right-hand-side pattern. This is in contrast to DDD, where each instance of an algebraic rewrite generated a proof condition for the external theorem prover, PVS [77].

Starfish's facilities for type management and identity application borrow from a large body of work known as *algebraic specification* [46, 32, 3]. In this approach to formal program development, specifications are defined by multisorted structures that are subject to axioms, while programs are defined by the *algebras* or *models* that satisfy the structure and axioms. *Stepwise refinement* [105, 90] is the process of embedding specifications—via structure preserving mappings—into successively more restrictive structures. These restrictions add details about data representation, algorithmic choices, interface methods, etc.

Several tools have been developed to promote the algebraic specification methodology. *CLEAR* [18] was the first language to support algebraic specification. *OBJ3* [31]

and later *Maude* [69] build on the CLEAR foundation by adding simulators, concurrency and a notion of objects. The *Larch Shared Language* [34] is an interface specification language that makes use of algebraic specifications with equational axioms and loose semantics. *Extended ML* [89] augments the standard ML type system with a facility for specifying behavioral requirements of code using axioms over higher-order logic and equality. *Common algebraic specification language (CASL)* [2] is a framework for algebraic specification and development which tries to integrate the best aspects of previous efforts and serves as a common platform for the deployment of algebraic techniques in subsequent languages.

Starfish is not a pure algebraic specification package. It combines the underlying stream semantics of DDD for process specification with algebraic specification techniques for term level manipulation. This hybrid approach to system specification has been used by the *language of temporal ordering specification (LOTOS)* [8]—an ISO-endorsed formal specification language for distributed systems, *evolving algebras* [33] which acts upon an algebra or model for a given structural signature with synchronous update rules—and CSP-CASL [87], which fuses the calculus of *communicating sequential processes (CSP)* [15] with the CASL system.

2.3 Stream modeling and transformational synthesis

The *evolving algebra* or *abstract state machine (ASM)* [33, 7] formalism bears several similarities with Starfish's behavior tables. The formalism applies guarded synchronous updates to a state represented as a multisorted algebra. The update may define a distinguished constant or the result of a function (of the algebra) applied to a specific input; for example, the expression $f(t_i, \ldots, t_n) = a$ assigns the value a to the output of f applied to the inputs $(t_i, \ldots, t_n)$. Subsequent applications of f respect this change. When updates are restricted to distinguished constants, they behave very much like registered signals in Starfish. The guards in evolving algebras are nested `if-then-else` statements keyed by boolean terms in the algebra. This style of update specification parallels selection guards in Starfish to determine signal updates. Just as compiler-machine derivations from language semantics motivated several DDD derivations, ASMs have formulated many compiler-machine specification and verification efforts. A formal definition of the Java language, compiler, and virtual machine is given with ASMs and then proven sound by rigorous mathematical discourse [94]; although the proof of compiler correctness spans 83 cases, they do not acknowledge any mechanical assistance with their argument.

Like DDD, O'Donnell's *Hydra* [75, 76] system describes hardware with recursive stream equations. Hydra develops the idea that one can produce meaningful alternative interpretations of a recursive circuit specification by redefining or *overloading* the primitive operations. One set of primitives simulates the circuit, while another

creates a *netlist* for fabrication, and yet another analyzes the circuit's critical path. Hydra is implemented in Haskell which has built-in facilities for equational reasoning. Hydra contains built-in integer-parameterized libraries for several classes of regular circuits (e.g., an n-bit adder).

Lava [20] is another functional hardware description language embedded in Haskell. Like Hydra, it overloads circuit primitives to achieve multiple interpretations from the same specification. Lava has well developed libraries for expressing regular connection patterns (e.g., tree and butterfly) that are common in digital signal processors. One Lava study specifies highly parameterized layout generators for multipliers based on (somewhat) regular tree connection structures [91]; parameters account for many low-level details that are invisible to other functional modeling disciplines including the wiring limits of target technologies and delay based on wire length. This focus on low-level parameterization led to the development of a relational description language, *Wired* [4], which provides accounts for all aspects of layout—including the precise location of wires. Generally, the relational model represents architecture as mathematical relations and architectural connection as relational composition. This approach is embraced by *Ruby* [92], an earlier language for describing synchronous circuits at the layout level. Also relevant to Starfish, Ruby developed a calculus for retiming and slowdown of circuits.

HAWK is another hardware description language embedded in Haskell that exploits overloaded primitives for alternate semantics. Hawk descriptions are at the register transfer level and higher, rather than the layout level. Hawk has successfully

specified superscalar microprocessors [21]. Hawk also includes a transformation algebra targeted at microprocessor architectures [66]. In particular, their retiming transformation is very similar to the one implemented by Starfish. Some general transformations have been implemented previously in DDD and other systems [82, 23], while the remaining ones are domain specific (concerning pipeline hazard and forwarding properties). They derived an unpipelined processor from a pipelined specification with their algebra. This simplified design became the subject of a verification effort. Their derivation was mechanized in the Isabelle [81] theorem prover.

The transformational approach to synthesis occurs outside of the stream modeling contexts as well. The HASH [23] synthesizer builds on an *automaton theory* embedded in HOL to represent systems. This approach, and others within the same research group [24, 25], leverage high-level synthesis heuristics [19, 70] developed outside of their formal framework to guide the formal transformation process within HOL. The HASH system also implements a retiming transformation very similar to the one Starfish implements.

Teica, Radhakrishnan, and Vemuri [82] combine control-flow and data-flow graphs (CDFG) to represent systems. They investigate the use of formal transformations in the verification of high-level synthesis algorithms. Their register merge and split operators produce effects similar to Starfish's serialization capability.

2.4 Tables in system development

Behavior tables were first proposed by Rath, Tuna, and Johnson [86] as an extension of register transfer tables. Their formulation defines table semantics with state machines rather than stream systems. The proposal includes a mechanism for bounded indirection over the table's internal registers, a language feature that eliminates many sorts of state transition redundancy [98]. The original proposal also served as a platform to integrate Rath's *Interface Specification Language (ISL)* [85, 84, 83] which defines data exchange between processes with finite-state machine semantics. ISL introduces the notion of *protocol complement* which specifies behavior of the state machine's environment. Composing a behavior table with an off-the-shelf component required embedding the component's protocol complement into the tabular expression.

Starfish's implementation does not realize the full scope of this original proposal, but corresponds closely to Johnson's formulation in terms of stream systems [55]. This version neither includes language support for register indirection nor transformations for ISL environment insertion. Even without this support, behavior tables can still express *state indirection* [98]. Moreover, Starfish's serialization and type translation facilities can insert linear protocol environments. Johnson and Tsow [61] present a transformation algebra that operates directly on behavior tables rather than some embedding inside another system (e.g., DDD). Tsow and Johnson [96] make a case for implementing behavior tables directly by illustrating various system factorizations, including a garbage collector decomposition, in their tabular algebra. Later, they extend the scope of factorizations by introducing algebraic data refinement [97].

Factorizations on the refined system could then cut across the implementation types—e.g., separate counter and pointer registers from a heap.

Prior to Starfish, number of specification systems had adopted tables as *formal objects* [49]. The first engineering tools to incorporate tables and other diagrams as formal expressions appear in *requirements specification*—the first stage of software engineering. Clarity at this stage is critical to both computer scientists and problem domain experts. The *Requirements State Machine Language (RSML)* [64, 38] describes Mealy machines using *statecharts* [36, 37]—a hierarchical state-machine diagram schema—and AND/OR *decision tables* [78] to guard transitions. RSML has extensive tool support and has been used to specify requirements for the *Traffic Collision Avoidance System 2 (TCAS II)*, a complex process control system for aircraft collision avoidance. The specification was regularly reviewed by non-computer scientist experts, whose feedback helped develop the particulars of both the RSML language and specifications.

Software Cost Reduction (SCR) [44, 43], another formal method for requirements specification, expresses requirements as collections of tables. The method was developed to document the flight software for the Navy's A-7 aircraft. SCR reduced the A-7's prior software documentation—literally shelves of manuals—to a single 500 page document. Though clarified and compacted by tables, the original application of the SCR technique still depended upon English specification. Parnas—one of SCR's founders—continued to develop the method, formalizing precise syntax and semantics for 10 classes of table expressions in software documentation [78, 79]. The requirements specification technique has subsequently been deployed in several high

assurance areas ranging from avionics to nuclear power. SCR* [39, 41] is a software tool that supports the SCR specification method which includes an editor, simulator, consistency checker, and a property verifier. The researchers contend that tool support is a prerequisite for the practical application of formal methods [40, 80]— a thesis that motivates Starfish's development.

Tablewise [47, 48] formally specifies software with variants of Parnas's decision tables [78]. Although similar to SCR* in many respects, Tablewise generalizes decision variables to nonboolean types and does not decompose specifications into mode classes—a hierarchical control mechanism in SCR*. The Tablewise toolkit includes a table editor, a table consistency and completeness analyzer, and a code generation module. Subsequent work by Hoover, Guaspari and Humenn [49] extends decision tables with preconditions and recursive partitioning (a scheme for reducing redundancy). Their report defines a *function table* which joins a decision table with output behaviors. When Starfish's behavior tables are limited to combinational output signals, they are semantically similar to these function tables. Moreover, the report calls for merging decision tables with state—a central feature of Starfish's behavior tables.

More recently, a specialized table-driven language has been developed for specifying cache coherence protocols [93, 65]. Their tables encode Mealy machines where the row labels vary over states, the column labels vary over events (input values), and the entries encode state transition and output emission. Like developers of RSML, SCR, and Tablewise, the authors praise tabular representation for combining mathematical rigor with accessibility to less mathematically-inclined users. Although their specifications could have been embedded in a mechanical theorem prover, they rely

on a paper proof to argue the specification's correctness.

Chapter 3

Design Derivation

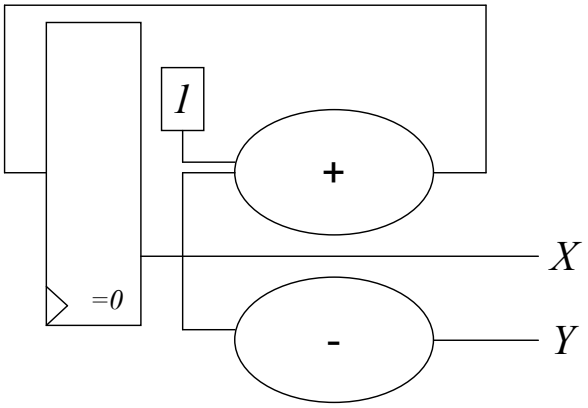
This chapter presents the underlying mathematical model for design derivation. The first section shows how stream equations model synchronous systems and relates stream equations to register-transfer diagrams. The next section develops the first-order type system that Starfish implements, beginning with a background of multisorted algebras and then proceeding to the system's type declarations, facilities for parameterization, and special types. The next section formally defines stream semantics. The last section examines definitions of equivalence and presents a transformational algebra for manipulating stream systems—including the five core transformations [54] that define the design space for DDD's architectural exploration and a sixth transformation which expands this space.

3.1 Modeling synchronous processes

Design derivation represents synchronous systems of computing processes as first-order mutually co-recursive stream equations. Below is a simple co-recursive system of equations, its solutions (streams over integers), and its schematic interpretation. The suffix * indicates the "lifting" of a term level function or constant to the stream

level: e.g., 1^* is a stream of 1's and $+^*$ is element-wise addition.

$$\begin{array}{lll|lll} X & = 0\,! & X +^* 1^* & X & = (& 0, \quad 1, \quad 2, ...) \\ Y & = & -^* X & Y & = (& 0, \quad -1, \quad -2, ...) \end{array} \tag{1}$$

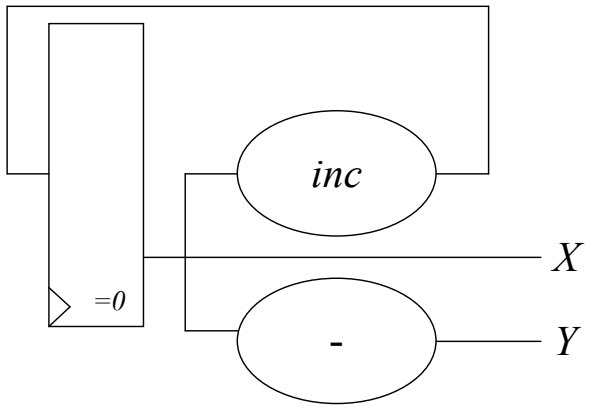


X and Y are variables over the stream of integers. These streams split into two parts: an integer head, and a stream-of-integers tail. The first equation specifies X's head as 0 and its tail as the stream-of-1's added element-wise to X. The second equation defines Y as the element-wise negation of X. These equations have unique solutions displayed on the right-hand side. The schematic interpretation corresponds to an abstract register-transfer system view. This interpretation motivates the name *signal* for variables in the equations. Signals defined by explicit specification of their head and tail (as with X) are *registered* or *sequential*. This naming reflects the use of the register to delay the increment of X to the next cycle. Signals defined as element-wise functions of other signals are *combinational*; e.g., Y, the negation of X, is not delayed by a register—its evaluation is visible in the present cycle.

There are many different systems of equations for any particular solution set. In the schematic interpretation, there are many architectures that produce the same

output streams. For instance, replacing a term by an algebraically equivalent term does not change the solution streams. If unary function $inc : integer \rightarrow integer$ is $\lambda x.x + 1$, then the following system has the same solution as (1):

$$\begin{array}{llll|lll} X & = 0\,! & inc^*(X) & & X & = (\ 0, & 1, & 2, ...) \\ Y & = & -^*X & & Y & = (\ 0, & -1, & -2, ...) \end{array}$$



(2)

This straightforward substitution changes the architecture. A potentially less complex one-input increment-function replaces the two-input adder. Johnson [54] summarizes a logic that manipulates architecture while maintaining the solution streams. The design derivation method is the intelligent guidance of this transformation algebra to produce system refinements.

3.2 Terms and types

The semantics of stream equations and the semantics of its constituent term expressions are orthogonal: the stream equations determine streams of uninterpreted terms, while term semantics maps streams-of-terms to streams-of-values. Applying the first

stage to equation (2) produces:

$$\begin{array}{lll|lll} X & = 0\,! & inc^*(X) & X & = (& 0, \quad inc(0), \quad inc(inc(0)), ...) \\ Y & = & -^*X & Y & = (& -0, \quad -inc(0), \quad -inc(inc(0)), ...) \end{array} \tag{3}$$

This section provides the framework for term syntax and semantics. It defines function and constant symbols, rules of composition, and semantic constraints through the declaration of term identities. The collection and enforcement of these rules constitute the *type system*.

3.2.1 Multisorted signatures and algebra

First-order multisorted algebra forms the mathematical foundation for the type system. A *signature*, denoted by Σ, prescribes term syntax. Σ is defined over a set of *sorts*, I. The signature consists of a set of symbols labeled by ordered tuples over I. For clarity, a symbol c labeled by a singleton i (denoted $c : i$) is a *constant* of sort i. The other symbols are *functions* from $i_1 \times \ldots \times i_{n-1}$ to i_n (written $f : i_1 \times \ldots \times i_{n-1} \rightarrow i_n$), when the sort label is $(i_1, \ldots, i_n)$. For the remainder of the text, "constant" and "function" replaces the mathematically concise labeled symbol. Σ terms have the following form:

- c is a *constant expression* of *type* i where i is the sort of c
- $f(t_1, \ldots, t_{n-1})$ is a *function application* of *type* i_n where $f : i_1 \times \ldots \times i_{n-1} \rightarrow i_n$ and t_m is an expression of sort i_m for $1 \leq m \leq n-1$.

Example 3.1: Let Σ be defined over a single-sort, *nat* with one constant, **0**, and one function $succ : nat \rightarrow nat$. This is the signature of the natural numbers.

Example 3.2: Let *atom* and *list* be sorts. To define a signature for a programmer's list facility, declare the following constants and signatures:

$$\begin{array}{ll} nil & : list \\ cons & : atom \times list \rightarrow list \\ car & : list \rightarrow atom \\ cdr & : list \rightarrow list \end{array} \tag{4}$$

A recursively defined *valuation function* specifies the semantics for Σ-expressions. For each sort i, let S_i be a *carrier set*. The valuation is higher order since it maps function symbols into function spaces. Square brackets $v[f]$ denote valuations applied to functions and rounded parentheses $v(t)$ denote valuations applied to terms. A valuation, v, is recursively defined:

For each constant $c : i$,

$$v(c) \quad := s \quad \text{where } s \in S_i$$

For each function $f : i_1 \times \ldots \times i_{n-1} \rightarrow i_n$,

$$v[f] \quad := \phi \quad \text{where } \phi \in S_{i_1} \times \ldots \times S_{i_{n-1}} \rightarrow S_{i_n}$$

For each function application $f(t_1, \ldots, t_m)$,

$$v(f(t_1, \ldots, t_m)) \quad := v[f](v(t_1), \ldots, v(t_m))$$

The valuation's third clause follows the form above, so it suffices to specify how v acts on the Σ's constants and functions. The *evaluation* of a term t is $v(t)$. The carrier sets together with v's mapping of the constant and function symbols form a Σ-*model*, more commonly known as a Σ-*algebra*.

Example 3.3: The standard algebra for $\Sigma_{nat} := (0 : nat, succ : nat \rightarrow nat)$ uses $\mathbb{N}$, the natural numbers, as its carrier set. It maps $0 : nat$ to $0 \in \mathbb{N}$ and *succ* to

$\lambda x.(x + 1)$.

Example 3.4: For Σ_{nat}, let $\{0, 1, 2\}$ be the carrier set, let $v(0 : nat) := 1$, and let $v[succ] := \{(0, 2), (1, 2), (2, 0)\}$. In this algebra, the expression $succ(0)$ evaluates to 2, the expression $succ(succ(0))$ evaluates to 0, and no application of *succ* evaluates to 1. While the signature of these terms is the same as the natural numbers, the semantics behave nothing like the natural numbers. The carrier set has only 3 elements, and the function *succ* has no particular property in common with the usual +1 definition of *succ*.

Identities constrain the space of algebras for a given signature Σ. The algebra must satisfy a set of universally quantified equations in the signature language. For each sort i, let V_i be a set of *variable names* such that $V_i \cap V_j = \emptyset$ when $i \neq j$. A variable in V_i has *type* or *sort* i. Let Σ^{func} and Σ^{const} be the sets of function and constant symbols for Σ. Identities and variable term expressions over Σ (*ident* and $varExp$ below) have the following form:

$$\begin{array}{lcl} ident & := & varExp = varExp \\ varExp & := & const \mid var \mid func(varExp \ldots) \\ func & \in & \Sigma^{func} \\ const & \in & \Sigma^{const} \\ var & \in & \bigcup_{i \in I} V_i \end{array} \tag{5}$$

such that the sorts of the function parameters correspond to the sort of the function signature.

Suppose a Σ-algebra has carrier sets C_i for $i \in I$ and a valuation v defining the mapping of constants and functions. An *assignment* α maps variables of a sort to carrier sets of a sort. Valuations extend to variable term expressions with assignments (written $v(\alpha, t_{varExp})$) by adding a variable term variant to the recursive definition in (6):

$$\begin{array}{lcl} v(\alpha, x) & := & \alpha(x) \quad \text{when } x \in \bigcup_{i \in I} V_i \\ v(\alpha, c) & := & v(c) \quad \text{when } c \in \Sigma^{const} \\ v(\alpha, f(t_1, \ldots, t_n)) & := & v[f](v(\alpha, t_1), \ldots, v(\alpha, t_n)) \end{array} \tag{6}$$

An algebra *satisfys an identity*, $\varphi = \psi$, when for every assignment α, $v(\alpha, \varphi) = v(\alpha, \psi)$.

Example 3.5: Extend the signature Σ_{nat} with the identity $succ(succ(y)) = y$ for variable symbol y. The standard natural numbers do not satisfy this identity because the assignment defined by $y \mapsto 1$ produces $3 \neq 1$. On the other hand, let $\mathbb{R}$ be the carrier set with $\mathbf{0} \mapsto 0 \in \mathbb{R}$ and $succ \mapsto \lambda x.(-x)$. Then for all assignments $y \mapsto r \in \mathbb{R}$

$$\begin{array}{lcl} v(\alpha, succ(succ(y))) & = & \lambda x.(-x)[v(\alpha, succ(y))] \\ & = & \lambda x.(-x)[\lambda x.(-x)[v(\alpha, y)]] \\ & = & \lambda x.(-x)[\lambda x.(-x)[r]] \\ & = & -(-r) \\ & = & r \\ & = & v(\alpha, y) \end{array} \tag{7}$$

thus satisfying the signature with identity.

Example 3.6: Recall the list signature $\Sigma_{list} = (nil : list, cons : atom \times list \rightarrow list, car : list \rightarrow atom, cdr : list \rightarrow list)$ from Example 3.2. We further stipulate that

all algebras must satisfy the following identities:

$$\begin{array}{rcl} car(cons(a,l)) & = & a \\ cdr(cons(a,l)) & = & l \end{array} \tag{8}$$

Let $\mathbb{Z}$ and the set stacks over integers, be the carrier sets for **atom** and **list**. The valuation is defined by:

$$\begin{array}{lcl} nil & \mapsto & emptyStack \\ cons & \mapsto & \lambda al.push(a,l) \\ car & \mapsto & \lambda l.top(l) \\ cdr & \mapsto & \lambda l.pop(l) \end{array} \tag{9}$$

where $pop(emptyStack) = emptyStack$ and $top(emptyStack) = 0 \in \mathbb{Z}$. Defined as above, the stack of integers satisfies the constructor-accessor identities for Σ_{list}.

3.2.2 Adapting the mathematics for use in Starfish

Starfish declares its types as a multisorted signature with identities. The signature declarations define the space of constant and function symbols, while the identities define the term level equational logic. Thus the type system specifies syntactic form and some semantic requirements. Term semantics are *loose*: there may be many algebras that satisfy the declarative structure and equational constraints, but Starfish does not commit to a particular algebra. As the derivation proceeds, the types are *refined* by homomorphic mappings into more tightly restricted types.

Starfish partitions the declaration of signatures over single-sorted declarations, parameterized signature declarations (i.e., signature schema), and inter-sort function declarations. Each one specifies a signature fragment with their "sum" prescribing

the overall system signature. In addition to the user specified types, Starfish provides extended support for two common system types: integers and bit vectors.

User defined signatures

The *unbounded single-sorted signature* declaration exactly matches the mathematical development in Section 3.2.1. A declaration contains a sort name, a list of constants, a list of functions, a list of variable names, and a list of identities. Since declarations are single-sorted, constant and variable sort tags are redundant and function signatures need only specify a number of inputs. Starfish does not supply a schema that specifies infinite sets of symbols, thus all signature components declare a finite (and usually small) number of symbols. Starfish uses the following syntax for unbounded single-sorted signature declarations:

```
(define-term-alg <name>
 (<const-sym> ...)
 ((<func-sym> <integer-arity>) ...)
 (<var-sym> ...)
 (('<ident-sym> <local-term-exp> <local-term-exp>) ...) )
```

where `<local-term-exp>` is an s-expression in the language of the constants, functions and variables inside the declaration.

Example 3.7: This is how Starfish declares group signature and axioms:

```
(define-term-alg group
 (id)
 ((op 2) (inv 1))
 (a b c)
 (('associative (op (op a b) c) (op a (op b c)))
  ('identity-left (op id a) a)
  ('identity-right (op a id) a)
  ('inverse (op a (inv a)) id)))
```

Enumerated types are a variant of single-sorted declarations. They share the same specification form but their semantics are further constrained. Valuations between constant symbols and carrier sets are bijections. There are no semantic values beyond those represented by the constant symbols. In practice, this facility defines states, data tags, and other control tags.

Example 3.8: Enumerated signature declarations can express booleans and their common identities:

```
(define-enum-alg boolean
  ;; list of constants
  (true false)
  ;; functions and arities
  ((conj 2) (disj 2) (inv 1))
  ;; identities over the variables x,y,z
  (x y z)
   ;; commutative laws
   (('conj-comm (conj x y) (conj y x))
    ('disj-comm (disj x y) (disj y x))
   ;; associative laws
    ('conj-assoc (conj (conj x y) z) (conj x (conj y z)))
    ('disj-assoc (disj (disj x y) z) (disj x (disj y z)))
   ;; Distributive laws
    ('conj-dist (conj x (disj y z)) (disj (conj x y) (conj x z)))
    ('disj-dist (disj x (conj y z)) (conj (disj x y) (disj x z)))
   ;; DeMorgan laws
    ('demorgan1 (inv (conj x y)) (disj (inv x) (inv y)))
    ('demorgan2 (inv (disj x y)) (conj (inv x) (inv y)))
   ;; Misc laws
    ('inv-def1 (inv true) false)
    ('inv-def2 (inv false) true)
    ('conj-def1 (conj true x) x)
    ('conj-def2 (conj false x) false)
    ('conj-def3 (conj x x) x)
    ('disj-def1 (disj true x) true)
    ('disj-def2 (disj false x) x)
    ('disj-def3 (disj x x) x)))
```

This signature has a unique algebra (up to isomorphism) because the carrier set is in one-to-one correspondence with the constant symbols. Alternatively, the above example could have enumerated the function tables without the aid of variables `x`, `y` and `z`:

```
...
('andff (and false false) false)
('andft (and false true) false)
('andtf (and true false) false)
('andtt (and true true) true)
...
```

The same unique algebra satisfies this definition, but such an approach prevents identity application to terms with variables.

Parameterized signatures declare a specific sort, *psort*, in terms of a finite set of *sort variables*, $sv_1, \ldots, sv_j$. We write this dependence $psort\{sv_1, \ldots, sv_j\}$. Starfish's declarative form for parameterized signatures extends the previous forms in two ways. They specify lists of sort variables and consequently must specify types (e.g., *psort*, sv_0, sv_1, ...) for functions and variables (constants have type *psort*).

```
(define-param-alg <name>
 (<sort-var-sym> ...)
 (<const-sym> ...)
 ((<func-sym> (<type-sym> ...) <type-sym>) ...)
 ((<var-sym> <type-sym>)...)
 (('<ident-sym> <local-term-exp> <local-term-exp>) ...) )
```

A paramaterized signature declaration is a schema. Schema instances specify concrete types for each sort variable.

Example 3.9: The following declaration specifies a *list* schema parameterized by the variable *atom*.

```
(define-param-alg list
 (atom)                    ; sort parameters
 (nil)                     ; constants
 ([cons (atom list) list] ; functions w/ explicit I/O
  [car (list) atom]
  [cdr (list) list])
 ([l list] [a atom])       ; typed formal variables
                           ; identities
 (['car-access (car (cons a l)) a]
  ['cdr-access (cdr (cons a l)) l]))
```

$list\{boolean\}$ is an instance of this parameterized signature. Its algebra has a *cons* function that maps a boolean and a list of booleans to a list of booleans; each reference to `atom` instantiates to `boolean`. In general a system signature may declare many (possibly nested) instances of the $list\{atom\}$ signature fragment; for example one term's type may be $list\{nat\}$, while another term in the same system expression is type $list\{list\{bool\}\}$.

While parameterized declarations promote reuse and modularity, they complicate uniqueness of symbols. Is the function symbol *cons* a mapping from $bool \times list\{bool\}$ to $list\{bool\}$ or a mapping from $nat \times list\{nat\}$ to $list\{nat\}$? Often there is a unique answer within the constraint of being "well-typed." Starfish employs a simple type inferencing algorithm to resolve the signature of every prameterized function and constant symbol. When the inferencer fails, the designer must disambiguate the types.

Integers and bit vectors

Starfish has special support for integers and bit vectors. The enhanced functionality includes: special constant recognition, limited facilities for evaluating constant expressions, and built-in identities. While it is theoretically possible to define the n-bit integers or vectors as a very large enumerated algebra, this task is infeasibly cumbersome.

Integer constants are decimal strings beginning with a nonzero digit and may be prefixed by a single $-$. Starfish evaluates constant integer expressions over the four operations $\{+.-, \times, \div\}$ according to the 2's complement encoding when $n \neq 0$ and according to the standard initial model when $n = 0$. Within a particular design, there is only one type of integers; e.g., there are not 32-bit and 64-bit integers in the same design scope. This is largely a tool implementation decision that removes ambiguity when parsing and evaluating integer expressions. One can still specify designs over "integers" of varying bit-lengths with the bit-vector facility.

Bit vectors are parameterized by integer constants (rather than other types). A derivation may include signals and terms of varying bit length; e.g., the SECD example in Section 9.2 uses 24-bit vectors for memory addresses and 32-bit vectors for data storage. Bit vector operators include the element-wise boolean functions $\{and, or, not, nand, xor\}$ as well as $\{lshift, rshift, parity\}$. Additionally, the designer may declare functions on bit-vectors that follow the semantics of alternative integer encodings (not just 2's complement).

Tuples and projectors

Tuples and their accessors are an important sort schema that cannot be declared with the `define-param-alg` facility. The schema has one constructor `tuple`, although Starfish denotes tuple construction with square brackets: e.g., `[1 2 false]` is short for `(tuple 1 2 false)`. The constructor accepts an arbitrary number of arguments with arbitrary type. Every input combination produces a different tuple sort. The accessors to the tuple sort are `1st`, `2nd`, `3rd`, etc.

Example 3.10: The `define-param-alg` signature declaration can define an ordered pair or 2-tuple:

```
(define-param-alg 2-tuple
 (t1 t2)                        ; parameter symbols
 ()                             ; constants
 ([2-tuple (t1 t2) 2-tuple]     ; functions w/ explicit I/O
  [1st (2-tuple) t1]
  [2nd (2-tuple) t2])
 ([tup 2-tuple] [a t1] [b t2]) ; typed formal variables
 (['1st-access (1st (2-tuple a b)) a]
  ['2nd-access (2nd (2-tuple a b)) b]))
```

Starfish's tupling facility is more general since it applies to any number of inputs.

Selectors

Starfish has polymorphic selectors that take an enumerated type or bit vector key, and select over uniformly typed branch terms. The order of branch selection for enumerated types corresponds to their order of declaration in `define-enum-alg`. When selecting over a bit vector, the order of branch selection goes from $0b0 \ldots 0.$ to $0b1 \ldots 1.$ Paramaterized types and unbounded term types are not valid key types in

a selector. Term semantics are completely functional in Starfish, so there are no side effects. Conditional expressions (`if`, for instant) often guard evaluation of branches by the predicate expression, so that exactly one branching expression is evaluated. This prevents side effects from evaluation of other branches. Since Starfish terms have no side-effects, applicative order evaluation suffices for selector terms.

Example 3.11: Suppose `state` is an enumerated type defined by the signature

```
(define-enum-alg state
 (wait add sub branch halt) ; constants
 () () ()) ; no functions or identities
```

Then for terms `t1, t2, t3, t4, t5` of all the same type, the following identities hold:

$$\begin{aligned} sel(wait, t1, t2, t3, t4, t5) &= t1 \\ sel(add, t1, t2, t3, t4, t5) &= t2 \\ sel(sub, t1, t2, t3, t4, t5) &= t3 \\ sel(branch, t1, t2, t3, t4, t5) &= t4 \\ sel(halt, t1, t2, t3, t4, t5) &= t5 \end{aligned} \tag{10}$$

Example 3.12: This next example shows the selector identities for the four constants in $bvec\{2\}$:

$$\begin{aligned} sel(0b00, t1, t2, t3, t4) &= t1 \\ sel(0b01, t1, t2, t3, t4) &= t2 \\ sel(0b10, t1, t2, t3, t4) &= t3 \\ sel(0b11, t1, t2, t3, t4) &= t4 \end{aligned} \tag{11}$$

Inter-signature function declarations and conditional identities

The declarative structures presented so far do not account for function signatures between arbitrary sorts. The `declare-funcs` form fills this need. The functions may optionally be parameterized. In case of parameters, the definition behaves as a schema, overloading the names of the functions. Unlike previous declarations where the only symbols "in scope" were those declared within itself, `declare-funcs` may reference any previously declared constant, function or type symbols including integers, bit vectors, tuples, projectors, and selectors. Term and type expressions appear in quotations marks, since the full term expression language makes slight departures from s-expression syntax to include type annotation for subterms and the square bracket tuple constructor. Figure 2 shows the full syntax for term and type expressions.

```
(define-param-alg <name>
 (<sort-var-sym> ...)
 ((<func-sym> (<type-str> ...) <type-str>) ...)
 ((<var-sym> <type-str>)...)
 (('<ident-sym> <term-str> <term-str>) ...) )
```

Example 3.13: As an example, consider these list utility functions parameterized over a type variable `atom`. There is an `empty-list` predicate and a function that measures the length of a list. Since the function signatures map between sorts that do not share local scope in previous structures, they cannot be declared elsewhere.

$$
\begin{array}{lcl}
termStr & = & annotatedTermStr \mid plainTermStr \\
annotatedTermStr & = & termStr\texttt{:}typeStr \\
plainTermStr & = & const \mid var \mid funcApp \mid sel \mid tuple \mid projector \\
constSym & \in & C \\
varSym & \in & V \\
funcApp & = & \texttt{(}funcSym\ termStr^{+}\texttt{)} \\
funcSym & \in & F \\
sel & = & \texttt{(sel}\ termStr\ termStr^{+}\texttt{)} \\
tuple & = & \texttt{[}termStr^{+}\texttt{]} \\
projector & = & \texttt{(}projSym\ termStr\texttt{)} \\
projSym & \in & \{\texttt{1st}, \texttt{2nd}, \texttt{3rd}, \texttt{4th}, \ldots\} \\
 & & \\
typeStr & = & termSigSym \mid enumSigSym \mid paramTypeStr \mid tupleTypeStr \\
paramTypeStr & = & paramSigSym\{typeStr^{+}\} \\
tupleTypeStr & = & \texttt{[}typeStr^{+}\texttt{]} \\
termSigSym & \in & T \\
enumSigSym & \in & E \\
paramSigSym & \in & P
\end{array}
$$

Figure 2: Grammar for type strings and term strings with type annotations. Let C, F and V be the constant, function and variable symbols in the scope of the subject string. Let T, E, and P be the sets of in-scope signature names for unbounded term signatures, enumerated signatures, and parameterized signatures.

```
(declare-funcs list-utils
  (atom);; var-type names
  ((nil? ("list{atom}") "boolean")
   (list-depth ("list{atom}") "integer"))
  ((l "list{atom}") (a "atom") (i "integer"))
  (('nil1 "(nil? nil:list{atom})" "true")
   ('nil2 "(nil? (cons a l))" "false")
   ('list-depth1 "(list-depth nil:list{atom})" "0")
   ('list-depth2 "(list-depth (cons a l))" "(+ (list-depth l) 1)")))
```

In this example, `list{atom}` annotates `nil` to resolve type ambiguity; e.g., `nil` could refer to stacks of integers instead of lists over the schematic variable `atom`.

Since identities in a `declare-funcs` clause range over selector terms, these blocks may also express conditional identities. Some identities have a small number of special

cases that prevent their declaration in `define-term-alg`, `define-enum-alg`, and `define-param-alg`, since identities in these contexts may use only locally declared symbols.

Example 3.14: Consider the identity $cons(car(l), cdr(l)) = l$. This identity only holds when l is not nil. The `sel` term enables a conditional form of this identity as follows:

$$sel(nil?(l), l, cons(car(l), cdr(l))) = l \tag{12}$$

Adding the following line to the `declare-funcs` identity block from Example 3.13 puts this conditional identity into Starfish.

```
('cons-car-cdr (sel (nil?  l) l (cons (car l) (cdr l))) l)
```

Starfish provides four declarations for defining signatures. Taken as a collection of sort tags, labeled constants, function signatures, and identities, they form the *specification signature*. For a given signature, $\mathcal{E}[V]$ denotes the set of term expressions with variables in V. In particular, $\mathcal{E}[\emptyset]$ is the set of constant terms.

Example 3.15: The following signature and sort declarations are a specification signature for a stack calculator. The first declaration is an enumerated type for its input instructions. The next enumerated type is an instruction classification type. The function `inst-cat` from the `declare-func` block maps instructions to their instruction category; i.e., the four arithmetic tags map to `alu-op`. `define-param-alg` specifies the standard stack signature using `push`, `pop`, `top`, and `empty-stack`. The

`declare-func` block expresses the conditional identity for popping followed by pushing (labeled `'pop-push`). The signature for `alu` uses a 2 bit vector to specify arithmetic on its two input integers. `inst->op` decodes the instruction into a 2 bit vector.

```
(define-enum-alg stack-calc-inst
  (psh drp add sub mul div) () () ())

(define-enum-alg stack-calc-inst-cat
  (psh-op drp-op alu-op) () () ())

(define-param-alg stack
  (mem) ; parameterized over stack contents
  (empty-stack)
  ([top (stack) mem]
   [pop (stack) stack]
   [push (stack mem) stack])
  ([s stack] [a mem])
  (['push-top (top (push s a)) a]
   ['push-pop (pop (push s a)) s]))

(declare-funcs stack-calc-helpers
  (mem) ; parameterized over stack contents
  ([inst-cat ("stack-calc-inst") "stack-calc-inst-cat"]
   [inst->op ("stack-calc-inst") "bvec{2}"]
   [alu ("bvec{2}" "integer" "integer") "integer"]
   [mt? ("stack{mem}") "boolean"])
  ([a "integer"] [b "integer"] [d "mem"] [s "stack{mem}"])
  (['alu-add "(alu 0b00 a b)" "(+ a b)"]
   ['alu-sub "(alu 0b01 a b)" "(- a b)"]
   ['alu-mul "(alu 0b10 a b)" "(* a b)"]
   ['alu-div "(alu 0b11 a b)" "(/ a b)"]
   ['mt1 "(mt? empty-stack:stack{mem})" "true"]
   ['mt2 "(mt? (push s d))" "false"]
   ['pop-push "(sel (mt? s) s (push (pop s) (top s)))" "s"]))
```

The total signature includes the special built-in sorts `bvec`, `integer`, and `boolean` two user defined enumerated types `stack-calc-inst` and `stack-calc-inst-cat`, and one parameterized schema `stack` that is instantiated as many times as necessary.

In Example 3.17's stack calculator specification below, there is only one instance, `stack{integer}`.

3.3 Stream Systems

Stream systems have a few more features than indicated by the examples in Section 3.1. First, stream systems accept a set of typed input streams $I_{var} \in I$. The stream system consumes one element from every input stream per step. Thus there are three kinds of signals: input, sequential and combinational, $I \cup S \cup C$ (these are disjoint sets of sort-labeled variables). The system specifies some subset of these signals as *observable* or as *output signals*, the remaining signals are *internal*. Finally, a sort-labeled metaterm character `#`, read *unspecified*, that serves as a *don't-care* value when used internally, and a *don't-know* character when it appears in the input stream. The syntax space in the presence of `#` is $\mathcal{E}[\{\#_\tau : \tau \textit{ is a sort label}\} \cup I \cup S \cup C]$. A stream system has the following form:

$$\begin{array}{lllll} \lambda \quad (I_{var_1} \ldots) = O_{var_1} \ldots & & & & \\ \quad \textit{where} & & & & \\ \qquad S_{var_1} & = & S_{ini_1} & ! & S_{tm_1} \\ \qquad \vdots & & & & \vdots \\ \qquad C_{var_1} & = & & & C_{tm_1} \\ \qquad \vdots & & & & \vdots \end{array} \tag{13}$$

where $I_{var_j} \in I$, $S_{var_j} \in S$, $C_{var_j} \in C$, $S_{ini_j} \in \mathcal{E}[\#]$, $S_{tm_j}, C_{tm_j} \in \mathcal{E}[\# \cup I \cup S \cup C]$ and $O_{var_j} \in I \cup S \cup C$

The principle of *co-recursion* uniquely defines functions into streams by specifying

the value of the stream's *head* (a stream element) followed by a function on the stream's *tail* (another stream). It justifies this thesis' formulation of stream system semantics. The following theorem states (without proof) Barwise and Moss's [5] expression of the co-recursion principle for streams. The *cons* operator, ! , explicitly shows a stream's head and tail.

Theorem 3.1 (Co-recursion Principle for Streams)**.** *Let C be an arbitrary set. Given functions $G : C \to A$ and $H : C \to C$, there is a unique function $F : C \to A^\infty$ satisfying the following, for all $c \in C$:*

$$F(c) = G(c) \ ! \ F(H(c)) \tag{14}$$

This formula is called the co-recursion equation *for F.*

Example 3.16: Let $f : \tau \to \tau'$ be a term function. The *lifted stream function* $f^* : \tau^\infty \to \tau'^\infty$ which applies f element-wise to stream elements is co-recursively defined by

$$f^*(s) = f(head(s)) \ ! \ f^*(tail(s)) \tag{15}$$

Definition 3.1. Let C_{var} *depend* on D_{var}, written $D_{var} \prec C_{var}$, when D_{var} is a subterm of C_{tm}. A system has *combinational feedback* when there is a cycle of dependence among combinational signals: $C_{var} \prec D_{var} \prec \ldots \prec C_{var}$.

Definition 3.2. A system is *well-typed* when every sequential signal variable has the same type as its initial value and update term—i.e., $\tau(S_{var}) = \tau(S_{ini}) = \tau(S_{tm})$—and

every combinational variable has the same type as its update term—i.e., $\tau(C_{var}) = \tau(C_{tm})$.

A *well-formed* stream system is well-typed, devoid of combinational feedback, and limited to combinational or sequential equations. Let

$$sub : \mathcal{E}[\# \cup I \cup S \cup C] \rightarrow \mathcal{E}[\# \cup I \cup S]$$

recursively eliminate references to combinational variables. For constants, input variables, sequential variables, and unspecified terms, *sub* is the identity. For combinational variables and function applications:

$$\begin{array}{rcl} sub[C_{var}] & = & sub[C_{tm}] \\ sub[f(t_1, \ldots, t_n)] & = & f(sub[t_1], \ldots, sub[t_n]) \end{array} \tag{16}$$

This function is defined on all combinational terms of a well-formed stream system because there is no combinational feedback.

Semantics for the stream system in (13) are defined as follows: Let $\tau_{I_j}^{\infty}$ for $1 \leq j \leq l$ be the type of the input streams. Let τ_{S_j} for $1 \leq j \leq n$ be the type of the terms S_{var_j}, S_{ini_j}, and S_{tm_j}. Let τ_{C_j} for $1 \leq j \leq m$ be the type of the terms C_{var_j} and C_{tm_j}. Let $Tr^* : \tau_{I_1}^{\infty} \times \ldots \times \tau_{I_l}^{\infty} \times \tau_{S_1} \times \ldots \times \tau_{S_n} \rightarrow (\tau_{S_1} \times \ldots \times \tau_{S_n} \times \tau_{C_1} \times \ldots \times \tau_{C_m})^{\infty}$ be the *signal trace function* for the stream system in equation (13) which maps input streams and a current state to a stream over the sequential and combinational signal types. The defining co-recursive equation denotes repeated expressions $E_0, \ldots, E_n$ by the expression $\overbrace{E_i}^{i/n}$.

$$Tr^*(\overbrace{I_i}^{i/l}\overbrace{s_j}^{j/m}) = (\overbrace{s_j}^{j/m}, \overbrace{sub(C_{tm_k})[\overbrace{I_{var_i}/head(I_i)}^{i/l}, \overbrace{S_{var_j}/s_j}^{j/m}]}^{k/n}) \\ \overbrace{\quad ! \; Tr^*(\overbrace{tail(I_i)}^{i/l}, S_{var_j}[\overbrace{I_{var_i}/head(I_i)}^{i/l}, \overbrace{S_{var_j}/s_j}^{j/m}, \overbrace{sub(C_{tm_k})[\overbrace{I_{var_i}/head(I_i)}^{i/l}, \overbrace{S_{var_j}/s_j}^{j/m}]}^{k/n}]]}^{j/m} \tag{17}$$

The stream system (13) produces the signal trace for input streams $\overbrace{I}^{i/l}$ as defined by $Tr^*(\overbrace{I}^{i/l}, \overbrace{S_{ini_i}}^{i/l})$. When the system is labeled *Sys*—(13) is an anonymous component—$Tr^*[Sys]$ denotes the trace of all input, combinational and sequential signals; $Tr^*_X[Sys]$ denotes the trace of signal X; $Tr[Sys] \stackrel{\text{def}}{=} Tr^*_O[Sys]$, the trace of the input and output signals, defines the *observable trace* of *Sys*.

The signal trace function maps a set of input streams and beginning state into a single stream over tuples that contain one element for every internal signal. Conceptually, it may be more natural to think of the trace as a collection streams (one for each internal signal). However, bundling the signals into a tuple term makes Tr^*'s definition easier to state because the co-recursion principle uniquely defines functions into a single stream space, not collections of streams.

Example 3.17: Equation (18) below shows a stack calculator specification using the type signature of Example 3.15. Its first input channel, *instr*, is an instruction token in the enumerated type $\{psh, drp, add, sub, mul, div\}$. The other input channel supplies a stream of integers. Internally, the calculator maintains an abstractly

specified operand stack over the integers. The function $instCat$ classifys the incoming instructions as an element of the enumerated type $\{pshOp, drpOp, aluOp\}$. An ALU instruction consumes the first two elements of the stack, and places the result on the twice popped stack. The function $alu(opCode, a, b)$ performs the arithmetic operation according to the value of $opCode$, while $instOp$ (written `inst->op` in the declaration) extracts the ALU operation code from the incoming instruction. This calculator makes no provisions for incorrect command sequences—e.g., stack underflow or division by 0. The system of stream equations follow:

$$\begin{array}{l} StackCalc(instr, a) = res \\ \quad where \\ \quad\quad s \;=\; push(mt, 0)\ ! \\ \quad\quad\quad\quad sel(instCat(instr), push(s, a), pop(s), \\ \quad\quad\quad\quad\quad\quad push(pop(pop(s)), alu(instOp(instr), top(s), top(pop(s))))) \\ \quad\quad res \;=\; top(s) \end{array} \tag{18}$$

Assume the standard term semantics for integers, stacks, and the expected interpretation for $alu : inst \times integer \times integer \rightarrow integer$. The signal trace function is defined by

$$\begin{array}{l} Tr^*(I, A, s) = (s, top(s))\ ! \\ \quad\quad\quad\quad Tr^*(tail(I), tail(A), \\ \quad\quad\quad\quad\quad\quad sel(instCat(head(I)), push(s, head(A)), pop(s), \\ \quad\quad\quad\quad\quad\quad\quad\quad push(pop(pop(s)), alu(instOp(head(I), top(s), top(pop(s)))))) \end{array} \tag{19}$$

Applying Tr^* with an initial state of $push(mt, 0)$ and a specific set of input streams (shown below) yields the following partial trace:

$$\begin{array}{lcl} I & = & (push, push, push, add, drop, \ldots) \\ a & = & (5, 7, 3, 1, 1, \ldots) \\ s & = & (\{0\}, \{0,5\}, \{0,5,7\}, \{0,5,7,3\}, \{0,5,10\}, \{0,5\}, \ldots) \\ res & = & (0, 5, 7, 3, 10, 5, \ldots) \end{array} \tag{20}$$

3.4 An algebra for system refinement

The intuitive example from Section 3.1 (p. 23) shows two stream systems that capture the same behavior. They are linked by the observation that $inc(x) = x + 1$. Design derivation generates correct system using a set of correctness preserving transformations like the one above. This section explores standards of correct implementation using the framework put forth by Aagaard et al. [1]. The five architectural transformations that form the underpinnings of prior implementations are presented, followed by a sixth transformation that is new to the algebra.

3.4.1 Correctness and trace comparison

Correctness in hardware synthesis is often determined by comparing specification and implementation traces—i.e., the solution streams to the respective systems of equations. In particular, the observable traces (p. 44) are the input streams and the output streams, but not the internal signals. In the stack calculator—Example 3.17—*instr*, a and *res* are observable, while s is an internal signal. For the purposes of correctness, the collection of values at a particular *trace index* are the subjects of comparison. The observable values for *StackCalc* at index 0 are $\{push, 5, 0\}$.

Aagaard et al. have proposed a classification matrix for correctness statements

concerning superscalar microprocessors [1]. The four axes of their taxonomy are *alignment*, *match*, *specification execution determinism*, and *implementation execution determinism*. All expressions in this thesis have deterministic execution leaving only differences in the first two properties.

Alignment is the method specifying which trace indices to compare. In general, it is a relation between specification and implementation trace indices. *Pointwise* alignment is the identity function, comparing every index of the specification to the same index in the implementation. A *stuttering* alignment is function that is increasing (i.e., $spec_i < spec_j$ implies $align(spec_i) < align(spec_j)$). These two suffice for designs generated by Starfish, although more involved relations based on pipeline flush points, instruction fetch and retire are common in microprocessor verification.

Match is the method comparing values determined by alignment. Equality is the simplest: two values match if and only if they are equal. Containment is slightly more general: every term t is *contained* by the metaterm `#`, written $t \subset \texttt{\#}$. The definition builds recursively over term variants; $t \subseteq s$ when $t = s$, or $s = \texttt{\#}$ and $\tau(t) = \tau(s)$, or

$$\begin{array}{ll} t \text{ is a function, } f(t_1, \ldots, t_n) & s = f(s_1, \ldots, s_n), t_i \subseteq s_i, 1 \leq i \leq n \\ t \text{ is a selector, } sel(t_1, \ldots, t_n) & s = sel(s_1, \ldots, s_n), t_i \subseteq s_i, 1 \leq i \leq n \\ t \text{ is a tuple, } [t_1, \ldots, t_n] & s = [s_1, \ldots, s_n], t_i \subseteq s_i, 1 \leq i \leq n \\ t \text{ is a projector, } \pi_i(t') & s = \pi_i(s'), t' \subseteq s' \end{array} \tag{21}$$

The match procedure may also translate data representation. For example, a specification may define internal signals over abstract stacks, while its implementation represents unobservable stack signals with a memory and pointer.

Starfish derives correct systems through a sequence of transformations. Each transformation preserves correctness according to a particular alignment and match.

The straightforward transformations are correct along pointwise alignment and term equivalence (with respect to sort identities). Some instantiate `#`, weakening the match to containment over equivalence. Serialization stretches the alignment to stuttering, while data refinement methods change the match.

3.4.2 Five correctness preserving transformations

Johnson [54] presents five correctness preserving rules (under pointwise alignment) for transforming stream systems. These rules and the formation of named combinations constitute an algebra for architectural refinement. While lacking a formal completeness or succinctness statement, these rules powered the following 15 years of design derivation results including multiple hardware synthesis case studies. This section reviews these first transformations, setting the stage for a sixth transformation that extends the design space. They are the basis for the behavior table algebra in the next chapter. Each of these transformations preserves alignment—i.e., compares systems pointwise. All but one matches using equivalence, while the last drops to containment over equivalence.

The transformations are formulated to exclude mal-formed systems in the results. In particular, *the transformations are only valid when their results are well-formed.*

Transformation 1 (Signal Introduction and Elimination)**.** *The single equation $Y = T$ may be added to (or deleted from) a system description under the conditions 1 (or 2):*

$$\begin{array}{l} SD(v,V) = Z \text{ where} \\ \quad \begin{array}{rcl} X_1 & = & S_1 \\ & \vdots & \\ X_n & = & S_n \end{array} \end{array} \quad \begin{array}{c} \overset{(1)}{\Rightarrow} \\ \overset{(2)}{\Leftarrow} \end{array} \quad \begin{array}{l} SD(v,V) = Z \text{ where} \\ \quad \begin{array}{rcl} X_1 & = & S_1 \\ & \vdots & \\ X_n & = & S_n \\ Y & = & T \end{array} \end{array} \tag{22}$$

For signal introduction (1), Y must be distinct from $\{V, X_1, \ldots, X_n\}$ and must introduce no combinational feedback. For signal elimination (2), the variables in Y must not occur in any defining expression other than T.

This transformation eliminates unnecessary signals or adds signals in a well formed manner. The resulting system in both cases is correct using pointwise alignment and equals match.

Transformation 2 (Combinational identification)**.** *Combinational signal identifiers can be exchanged with their defining expressions wherever these occur. Substitutions may not introduce combinational feedback.*

$$\begin{array}{rcl} & \vdots & \\ X & = & S \\ Y & = & T \\ & \vdots & \end{array} \quad \Leftrightarrow \quad \begin{array}{rcl} & \vdots & \\ X & = & S[T/Y] \\ Y & = & T \\ & \vdots & \end{array} \tag{23}$$

This is a basic substitution principle stemming from the equations. Prior implementations limit this kind of substitution to subexpressions defined in combinational expressions. For example:

$$\begin{array}{rcl|rcl} X & = 1\,! & inc(X) & X & = (\ 1, & 2, \quad 3, ...) \\ Y & = & -X & Y & = (-1, & -2, \ -3, ...) \\ Z & = 1\,! & Z \times Y & Z & = (\ 1, & -1, \quad 2, ...) \end{array} \tag{24}$$

is equivalent to

$$\begin{array}{lll|lll} X & = 1\,! & inc(X) & X & = (\ 1, & 2, & 3, ...) \\ Y & = & -X & Y & = (\ -1, & -2, & -3, ...) \\ Z & = 1\,! & Z \times (-X) & Z & = (\ 1, & -1, & 2, ...) \end{array} \tag{25}$$

using combinational substitution as prescribed by Y.

An equation where a variable reference expands into a sequential expression (i.e., a head value followed by a co-recursively defined tail) does not generally simplify to either a sequential or combinational equation. Consider expanding the variable X in the combinational signal Z:

$$\begin{array}{lll|lll} X & = 1\,! & inc(X) & X & = (\ 1, & 2, & 3, ...) \\ Y & = 1\,! & Y \times 2 & Y & = (\ 1, & 2, & 4, ...) \\ Z & = & X \times Y & Z & = (\ 1, & 4, & 12, ...) \end{array} \tag{26}$$

expands to

$$\begin{array}{lll|lll} X & = 1\,! & inc(X) & X & = (\ 1, & 2, & 3, ...) \\ Y & = 1\,! & Y \times 2 & Y & = (\ 1, & 2, & 4, ...) \\ Z & = & (1\,!\, inc(X)) \times Y & Z & = (\ 1, & 4, & 12, ...) \end{array} \tag{27}$$

The resulting equation for Z does not simplify to a combinational or sequential form. Alternatively, one could have expanded the variable X within X's equation leading to two levels of delay in a single expression, $X = 1\,!\, inc(1\,!\, X)$. These non-combinational and non-sequential results motivated the decision to limit substitution to combinationally defined variables.

As with the previous transformation, the results are correct in the strongest possible way: pointwise alignment and equal match.

Transformation 3 (Grouping)**.** *A subsystem is grouped by nesting its identifiers and forming a tuple of their defining expressions.*

$$\begin{array}{rcl} & \vdots & \\ X_i & = & S_i \\ X_{i+1} & = & S_{i+1} \\ & \vdots & \\ X_{i+m} & = & S_{i+m} \\ & \vdots & \end{array} \quad \Leftrightarrow \quad [X_i\ X_{i+1}\ \ldots\ X_{i+m}] \begin{array}{c} \vdots \\ = \\ \vdots \end{array} [S_i\ S_{i+1}\ \ldots\ S_{i+m}] \tag{28}$$

This provides grouping and splittling of related signals. While this appears to be mere syntactic sugar, it extends signal introduction and elimination to sets of mutually dependent equations by bundling signals into tuple elements. The result is correct pointwise, but with a modified match. The matching criterion is satisfied when the elements of the tuple are equal to the values of the individual signals.

Transformation 4 (Replacement)**.** *Any term may be replaced by an equivalent one.*

$$\begin{array}{rcl} & \vdots & \\ X & = & R[S/Y] \\ & \vdots & \end{array} \quad \overset{S=T}{\Leftrightarrow} \quad \begin{array}{c} \vdots \\ X \\ \vdots \end{array} \begin{array}{c} \\ = \\ \\ \end{array} \begin{array}{c} \\ R[T/Y] \\ \\ \end{array} \tag{29}$$

This is a second substitution principle. Term level identities from signature declaration justify this transformation's expression replacements. It's results are pointwise correct using algebraic equivalence as the matching criterion.

Transformation 5 (Collation)**.** *If signal expressions S and T are of the same type, then X and Y can be combined over a compatible selector.*

$$\begin{array}{rcl} & \vdots & \\ X & = & sel(P,S,\#) \\ Y & = & sel(P,\#,T) \\ & \vdots & \end{array} \quad \overset{\tau(S)=\tau(T)}{\Rightarrow} \quad \begin{array}{rcl} & \vdots & \\ X & = & sel(P,S,T) \\ Y & = & X \\ & \vdots & \end{array} \tag{30}$$

This transformation is really a combination of # instantiation and identification: instantiate # in X to T, instantiate # in Y to S, identify $sel(P,S,T)$ in Y as X. Again correctness employs pointwise alignment but uses containment as the matching property (hence a one-way transformation $\Rightarrow$). The resulting system implements one possible behavior of the original system. Unlike the previous transformations, one system implements the other but not vice versa.

3.4.3 Extending identification to sequential signals

As shown in (27) on page 50, unconstrained substitution of sequential signal definitions for sequential variable references can produce malformed systems. In some circumstances, expanding sequential definitions lead to well-formed systems. The equations below show the simultaneous identification of sequential variables X and Y followed by the distribution of $\times$ over the two sequential subexpressions. The result is a well formed system:

$$\begin{array}{rcl|rcl} X & = 1\,! & inc(X) & X & = (\ 1, & 2, \quad 3, ...) \\ Y & = 1\,! & Y \times 2 & Y & = (\ 1, & 2, \quad 4, ...) \\ Z & = & (1\,!\,inc(X)) \times (1\,!\,Y \times 2) & Z & = (\ 1, & 4, \ 12, ...) \end{array} \tag{31}$$

$$\begin{array}{lll|lll} X & = 1\,! & inc(X) & X & = (\ 1, & 2, \quad 3, ...) \\ Y & = 1\,! & Y \times 2 & Y & = (\ 1, & 2, \quad 4, ...) \\ Z & = 1 \times 1\,! & inc(X) \times (Y \times 2) & Z & = (\ 1, & 4, \quad 12, ...) \end{array} \tag{32}$$

This transformation turns a combinational signal into a sequential one. The resulting system is correct under pointwise alignment and equal matching. The following transformation formulates this example's transformation in a general context. It is essentially a form of the Identification rule in the previous section, but has not been used in design derivation prior to this.

Transformation 6 (Sequential Identification)**.** *Let I, X, Y be input, sequential and combinational signals, respectively. Let Z be a combinational signal dependent only on X. Let O be an output signal which is necessarily an internal signal. Then the following systems have the same solution sets:*

$$\begin{array}{l} Sys(I) = O \\ \quad where \\ \quad\quad X = X_0\,! \quad f(X,Y,Z,I) \\ \quad\quad Y = \quad\quad h(X) \\ \quad\quad Z = \quad\quad g(X,Y,I) \end{array} \quad \Leftrightarrow \quad \begin{array}{l} Sys(I) = O \\ \quad where \\ \quad\quad X = X_0\,! \quad f(X,Y,Z,I) \\ \quad\quad Y = h(X_0)\,! \quad h(f(X,Y,Z,I)) \\ \quad\quad Z = \quad\quad g(X,Y,I) \end{array} \tag{33}$$

Proof. We prove this co-inductively ([72] contains a full explanation of this technique) by showing that the stream trace functions are equal when initialized with the proper initial state. The trace functions for the left- and right-hand systems are:

$$\begin{array}{lcl} Tr^*_l(I,x) & = & (x, h(x), g(x, h(x), head(I)))\,! \\ & & \quad Tr^*_l(tail(I), f(x, h(x), g(x, h(x), head(I)), head(I))) \\ Tr^*_r(I,x,y) & = & (x, y, g(x, y, head(I)))\,! \\ & & \quad Tr^*_r(tail(I), f(x, y, g(x, y, head(I)), head(I)), \\ & & \quad\quad h(f(x, y, g(x, y, head(I)), head(I)))) \end{array} \tag{34}$$

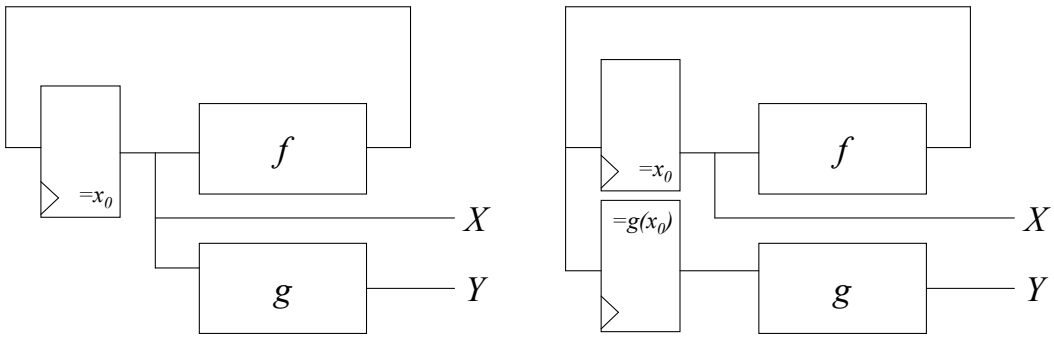


Figure 3: Combinational to sequential signal transformation on schematic representation of the system fragment, $X = x_0 \,!\, f(X), Y = g(X)$

For every input stream I and state x the heads are equal:

$$\begin{array}{rcl} head(Tr_l^*(I,x)) & = & (x, h(x), g(x, h(x), head(I)) \\ & = & head(Tr_r^*(I, x, h(x))) \end{array} \tag{35}$$

The tails are also equal by application of the co-inductive hypothesis between the first and second lines:

$$\begin{array}{rcl} tail(Tr_l^*(I,x)) & = & Tr_l^*(tail(I), f(x, h(x), g(x, h(x), head(I)), head(I))) \\ & = & Tr_r^*(tail(I), f(x, y, g(x, y, head(I)), head(I)), \\ & & \quad h(f(x, y, g(x, y, head(I)), head(I)))) \\ & = & tail(Tr_r^*(I, x, h(x))) \end{array} \tag{36}$$

□

Grouping extends this transformation to multi-input functions. One can always group all sequential signals to a single variable (X in this case) and combinational signals to another (Z) after eliminating dependencies by combinational substitution. The remaining signal Y depends only on sequential signals within the system.

Example 3.18: The stack calculator (Example 3.17) is a simple expression that characterizes behavior without regard to efficiency. At the very least, a target implementation needs a register to hold the stack's `top` value. Sequential identification produces this effect, however once the transformation takes place, the system no longer implies the combinational identity $res = top(s)$. So the designer must apply such combinational identifications before transforming *res* into a sequential signal. The following equations indicate modified terms with boldface:

First, replace instances of $top(s)$ by *res*

$$\begin{array}{l} StackCalc(instr, a) = res \\ \quad where \\ \quad\quad \begin{array}{lcl} s & = & push(mt, 0)\ ! \\ & & \quad sel(instCat(instr), push(s, a), pop(s), \\ & & \quad\quad push(pop(pop(s)), alu(instOp(instr), \mathbf{res}, top(pop(s))))) \\ res & = & top(s) \end{array} \end{array} \tag{37}$$

Next, transform *res* into a sequential signal. The application of *top* distributes over the selector's branches.

$$\begin{array}{l} StackCalc(instr, a) = res \\ \quad where \\ \quad\quad \begin{array}{lcl} s & = & push(mt, 0)\ ! \\ & & \quad sel(instCat(instr), push(s, a), pop(s), \\ & & \quad\quad push(pop(pop(s)), alu(instOp(instr), res, top(pop(s))))) \\ res & = & top(\mathbf{push}(\mathbf{mt}, \mathbf{0}))\ ! \\ & & \quad sel(instCat(instr), top(\mathbf{push}(\mathbf{s}, \mathbf{a})), top(\mathbf{pop}(\mathbf{s})), \\ & & \quad\quad top(\mathbf{push}(\mathbf{pop}(\mathbf{pop}(\mathbf{s})), \mathbf{alu}(\mathbf{instOp}(\mathbf{instr}), \mathbf{res}, \mathbf{top}(\mathbf{pop}(\mathbf{s})))))) \end{array} \end{array} \tag{38}$$

Finally, apply instances of $top(push(s, d)) = d$ to simplify terms.

$$
\begin{array}{l}
StackCalc(instr, a) = res \\
\quad where \\
\quad\quad s \;=\; push(mt, 0)\ ! \\
\quad\quad\quad\quad sel(instCat(instr), push(s, a), pop(s), \\
\quad\quad\quad\quad\quad\quad push(pop(pop(s)), alu(instOp(instr), res, top(pop(s))))) \\
\quad\quad res \;=\; 0\ ! \\
\quad\quad\quad\quad sel(instCat(instr), \mathbf{a}, \mathbf{top(pop(s))}, \\
\quad\quad\quad\quad\quad\quad \mathbf{alu(instOp(instr), res, top(pop(s)))})
\end{array}
\tag{39}
$$

3.4.4 Soundness of fold and unfold transformations

Fold and unfold transformations, part of a system of program transformations introduced by Burstall and Darlington [17], alter recursive function definitions by replacing function calls with instantiated function bodies and vice versa. Arbitrary applications of fold and unfold recursive function definitions can weaken the definition so that it no longer specifies a unique function. David Sands work on transforming functional programs [88] illustrates this phenomenon with the following example:

$$
\begin{array}{rcl}
f(x) & \stackrel{\text{def}}{=} & x + 42 \\
f(0) & = & 0 + 42 \;=\; 42 \\
\hline
f(x) & \stackrel{\text{def}}{=} & x + f(0)
\end{array}
\tag{40}
$$

One applies the *primitive law*, $42 = 0 + 42$, and then folds $0 + 42$ into a call to f on 0. This fold operation has introduced recursion and weakens the definition. Any linear function of the form $f(x) \stackrel{\text{def}}{=} x + c$ satisfies the resulting recurrence relation.

Two of Starfish's transformations, combinational (p. 49) and sequential identification (p. 53), are fold/unfold transformations on streams. The replacement rule (p. 51) parallels the application of primitive laws in Burstall and Darlington's system. Starfish's stream system transformations are all contingent on the well-formedness (p. 43) of the resulting systems. Well-formedness—i.e., all equations are combinational or sequential, all equations are well-typed, and no combinational cycles exist—ensures unique solutions by the co-recursion principle for streams (p. 42). In particular, unconstrained folds and unfolds are not allowed because they can introduce combinational cycles or create equations that are neither combinational or sequential (as shown in Section 3.4.3, p. 52).

Chapter 4

Behavior Tables

Systems of co-recursive stream equations and its transformational logic are the formal backbone of design derivation. These equations quickly become large, though well structured, often spanning several pages to describe even moderately sized systems. Realizing the need for a more comprehensible presentation, the developers of DDD adopted an informal tabular notation that reduces pages of co-recursive equations to a perspicuous 2-dimensional form. In practice, these tables guided their derivation strategy more than the equations themselves. This chapter formalizes a tabular notation of stream systems by presenting their syntax, semantics, and transformation logic. *Behavior table* display and the execution of their algebra are the principle functions of Starfish.

4.1 Behavior Table Expressions

Behavior tables are closed expressions composed of first order terms over user specified algebraic structures. As with co-recursive stream equations, variables are input I, sequential S or combinational C. Term evaluation follows the scheme presented in Section 3.2 (p. 24). A behavior table has the form:

Name: *Inputs* → *Outputs*	
Conditions	*Registers and Signals*
⋮	⋮
Guard	*Computation Step*
⋮	⋮

(41)

Inputs is a list of input variables and *Outputs* is a subset of the sequential and combinational variables. *Conditions* is a set of terms denoting values that range over enumerated types and bit-vectors. The *guards* are tuples of constants denoting values that the conditions can take. The conditions together with the guards form a *decision table*. Each guard indexes a *computation step* or *action*. An action is tuple of terms, each corresponding to a combinational or sequential variable. The actions and the internal signal names (i.e. non input variables) compose the *action table*.

There is a precise mapping between behavior tables and stream equations that fully captures behavior table semantics. Transforming a system of equations begins with choosing a uniform selector for all equations. For example, let $X = sel(k, a, b)$ and $Y = sel(l, c, d)$ be two equations in the system.

$$sel_{k,l} = \lambda(x_1, x_2, x_3, x_4).sel(k, sel(l, x_1, x_2), sel(l, x_3, x_4)) \tag{42}$$

is a composition of the two selectors that evaluates conditions k and l sequentially. Semantically, the key does not guard evaluation of branches (as the *if* statement of the Scheme programming language). The two equations may be rewritten with the selector composition: $X = sel_{k,l}(a, a, b, b)$ and $Y = sel_{k,l}(c, d, c, d)$. Assuming that l and k range over bits, a behavior table characterizing these two equations uses k

and l as the condition headings for the decision table, while the branches determine simultaneous updates (or actions) to X and Y:

Table Fragment for X and Y			
k	l	X:comb.	Y:comb.
0	0	a	c
0	1	a	d
1	0	b	c
1	1	b	d

(43)

The following example illustrates this transformation on the stack calculator expression from Example 3.17.

Example 4.1: The stack calculator from Example 3.17 (p. 46) is shown below:

$$
\begin{array}{l}
StackCalc(instr, a) = res \\
\quad where \\
\quad\quad s \;\; = push(mt, 0) \;!\; sel(instCat(instr), push(s, a), pop(s), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad push(pop(pop(s)), alu(instr, top(s), top(pop(s))))) \\
\quad\quad res = \; top(s)
\end{array}
$$

First we expand left hand side of $res = top(s)$ to use the same selector as the equation for s. The result is a system that uses homogeneous selectors:

$$
\begin{array}{l}
StackCalc(instr, a) = res \\
\quad where \\
\quad\quad s \;\; = push(mt, 0) \;!\; sel(instCat(instr), push(s, a), pop(s), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad push(pop(pop(s)), alu(instr, top(s), top(pop(s))))) \\
\quad\quad res = \; sel(instCat(instr), top(s), top(s), top(s))
\end{array}
\tag{44}
$$

A behavior table aligns signal updates in rows according to selector branches. In this case there is only one level of selection that ranges over the enumerated type $StackCalcInstCat$. The values $pshOp$, $drpOp$ and $aluOp$ index the possible signal

updates. In every case, the combinational signal update *res* remains *top*(*s*). The action table columns hold the signal updates. Column headings indicate signal kind (sequential or combinational), and the table heading indicates input and output signals.

StackCalc(instr,a) = res		
instCat(instr)	s:Seq	res:Comb
pshOp	push(s,a)	top(s)
drpOp	pop(s)	top(s)
aluOp	push(pop(pop(s)),alu(instr,top(s),top(pop(s))))	top(s)

(45)

Like stream systems, behavior tables denote communicating processes, rather than sub-procedures. Consequently behavior tables cannot themselves be entries in other behavior tables. Instead, they are composed by interconnecting I/O ports—*instr*, *a* and *res* in the case of (45). A *connection map* that is faithful to each component's arity specifies the composition. In the function-oriented modeling notation, such compositions are expressed as named recursive systems of equations,

$$\begin{array}{l} \mathsf{S}(U_1,\ldots,U_n) = (V_1,\ldots,V_m) \textit{ where} \\ \quad \begin{array}{rcl} (X_{11},\ldots,X_{1q_1}) & = & \mathcal{T}_1(W_{11},\ldots,W_{1\ell_1}) \\ & \vdots & \\ (X_{p1},\ldots,X_{pq_p}) & = & \mathcal{T}_p(W_{p1},\ldots,W_{p\ell_p}) \end{array} \end{array} \tag{46}$$

in which the defined variables X_{ij} are all distinct, each $\mathcal{T}_k$ is the name of a behavior table or other composition, and the outputs V_k and internal connections W_{ij} are all simple variables coming from the set $\{U_i\} \cup \{X_{jk}\}$.

4.2 Behavior Table Algebra

This section casts the transformations presented in Section 3.4.2 (p. 48) in terms of behavior tables and their hierarchical composition. Like their stream-oriented counterparts, the table algebra applies to architectural refinement. As before, this set is not claimed to be complete nor is minimal in any mathematical sense. Rather, this set of rules need only be robust enough to serve as a core rule set for tool implementation.

4.2.1 Notational conventions

Defining these rules has led to some challenging notational issues. To reduce clutter, we consider some novel conventions for expressing features, particularly for quantification. For reasons of both typography and clarity, we want to limit use of ellipses, columns, and subscripts to describe a table as, for example,

	$b\colon (I_1, \ldots, I_k) \to (O_1, \ldots, O_\ell)$					
	P_1	$\cdots$	P_m	S_1	$\cdots$	S_p
1	g_{11}	$\cdots$	g_{1m}	t_{11}	$\cdots$	t_{1p}
	$\vdots$	$\ddots$		$\vdots$	$\ddots$	
n	g_{n1}		g_{nm}	t_{n1}		t_{np}

(47)

The *table-scheme* notation uses the table itself as a quantifier, and uses set elements as indexes rather than number ranges. Uppercase italic variables denote sets; and differently named sets are always assumed to be finite and disjoint. Lowercase *italic* variables denote indices ranging over sets of the same name. The form

$$\begin{array}{c} S \\ R \boxed{x_{rs}} \end{array} \tag{48}$$

represents a two-dimensional array (table) of items, $\{x_{rs} \mid r \in R \text{ and } s \in S\}$. A sans seriff identifier denotes a fixed (throughout the scope of the rule) element from the set of the same name. Thus, the form

$$\begin{array}{c} \mathsf{s} \\ R \boxed{x_{r\mathsf{s}}} \end{array} \tag{49}$$

represents a column, $\{x_{r\mathsf{s}} \mid r \in R\}$ and similarly for rows.

Under these conventions, the table-scheme from Section 4.1 looks like

	b: I → O	
	P	*S*
1 ⋮ N	$g_{n,p}$	$t_{n,s}$

(50)

The use of ellipses $1 \cdots N$ on the left is not necessary, but serves as an reminder that the rows are typically numbered.

4.2.2 The rules

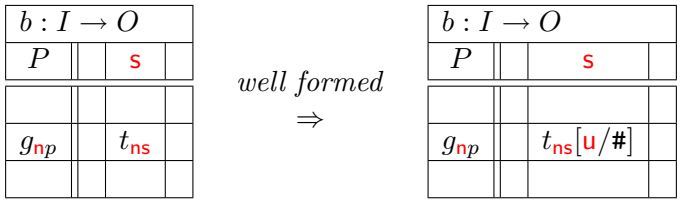
This section develops table-oriented counterparts to the stream-oriented rules of Section 3.4 (p. 46). Some rules effectively define table semantics by asserting equivalence between table structures and term expressions. This is most evident with the interplay between term selectors and the decision table. Similarly, the action grouping and ungrouping presents the columns in the action table in terms of polymorphic

tuples. The algebra of hierarchical lexical scoping is an assumption of the equational presentation that we make more concrete here with the decomposition and flattening rules.

Some structural rules subsumed by the semantics are not presented in this section, but should be supported by a full implementation. For example, permuting columns within the decision table is valid due to the commutativity and associativity of selector composition. Similarly the ordering of rows (guarded actions) and action table columns is a property of expression layout and irrelevant to evaluation. Their order may consequently be arbitrarily permuted. Also, *renaming* variables is allowed under the usual rules of α-substitution [27].

All transformations preserve pointwise alignment (p. 47). The match predicate is term equivalence (e.g. modulo the term identities specified by data types) in all cases except for subterm instantiation of output signals which uses containment.

Subterm Instantiation



(51)

Unspecified subterms, #, of the action table may be arbitrarily instantiated to u subject to the constraints of well formedness—i.e., identifiers must be in scope and the subterm may not introduce combinational feedback.

Signal introduction

$b : I \to O$	
P	S
g_{np}	t_{ns}

y *fresh*
well formed
$\Rightarrow$

$b : I \to O$		
P	S	y
g_{np}	t_{ns}	$\#_{n\mathsf{y}}$

(52)

A new combinational or sequential signal of any type may be added. The initial action values are typed #s. Together with subterm instantiation, this rule enables the introduction of arbitrary well formed signals and actions to a behavior table; this parallels the forward direction of Transformation 1 (p. 48).

Signal Elimination

$b : I \to O$		
P	S	R
g_{np}	t_{ns}	u_{nr}

$r \in R$ *unused*
$\Rightarrow$

$b : I \to O$	
P	S
g_{np}	t_{ns}

(53)

Eliminating a set of signals, R, is allowed when no r is a subterm of t_{ns}, and r is not an output signal. In summary, a set of signals may be removed when they are collectively unused by the residual system.

Combinational identification

$b : I \to O$			
P		s	y : comb
g_{np}		t_{ns}	r_{ns}

well formed
$\Rightarrow$

$b : I \to O$			
P		s	y : comb
g_{np}		$t_{\mathsf{ns}}[r_{\mathsf{ns}}/\mathsf{y}]$	r_{ns}

(54)

Combinational Identification substitutes the defining expression of a combinational signal for subterm instances of the signal's variable. Only substitutions which preclude combinational feedback are valid transformations.

Sequential identification

$b : I \to O$			
P	$S : t_n^{ini}$	$C : comb$	y : comb
g_{np}	t_{ns}	u_{nc}	$\mathsf{r}_{n\mathsf{y}}$

r *has no subterms in* I *or* C $\quad \Updownarrow$ (55)

$b : I \to O$			
P	$S : t_n^{ini}$	$C : comb$	y : $\mathsf{r}[s/t_s^{ini}]$
g_{np}	t_{ns}	u_{ns}	$\mathsf{r}[s/t_{ns}]_{n\mathsf{y}}$

Sequential identification is the tabular version of Transformation 6 (p. 53). The transformation requires identical action terms, r, throughout y's column, and that none of r's subterms is an input or combinational signal. Constraining y to have identical action terms (indicated by r) is an expressive convenience; repeated applications of the *decision instantiation* and *decision identification* rules (below) can always collapse the decision table to a single row, trivially satisfying term uniformity. Similarly, disallowing combinational subterms serves as a notational convenience; combinational variables may always be eliminated by repeated substitution of their action updates for their variable instances. The substantive constraint disallows access to input variables in r. Simultaneously substituting the defining actions for all the variables in r

results in an equivalent sequential signal representation of y as in Transformation 6. The initial value, represented in the column heading of sequential signals, is also the simultaneous substitution of the initial values into r.

Action grouping

$b : I \to O$			
P		G	
g_{np}		t_{ng}	

$g \in G$ *uniformly sequential or combinational* $\Leftrightarrow$

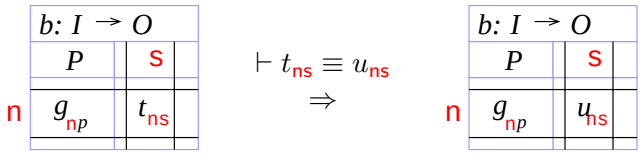
$b : I \to O$			
P		$G : comb$	g
g_{np}		$n^{th}(\mathrm{g})$	$[t_g]_n$

(56)

Columns can be grouped and ungrouped as long as the resulting columns are purely sequential or purely combinational. The behavior table form does not allow for structured signal identifiers. Grouping an ordered set of signals G produces a new signal g whose actions are ordered tuples corresponding to the actions of G. Remaining references to signals in G are resolved with combinational accessors to g.

This rule also characterizes ungrouping ($\Leftarrow$) in a general manner in context of the other transformations. If the goal is to ungroup an arbitrary tupled signal g, then one should first introduce a set of combinational signals G for each tuple component. The action columns are uniformly initialized to access the appropriate tuple component. In case an action update subterm $t_{n\mathrm{g}}$ to g is not a tuple constructor (e.g. the identity expression g), replace $t_{n\mathrm{g}}$ by the identical term $[\pi_1(t_{n\mathrm{g}}) \ldots \pi_{length(\mathrm{g})}(t_{n\mathrm{g}})]$. Now application of $\Leftarrow$ produces a suitable ungrouping.

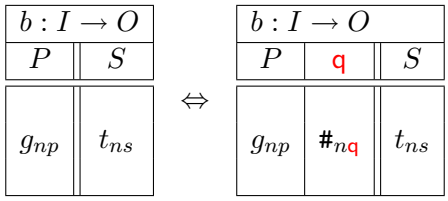
Replacement



(57)

An action table term can be replaced by another term that is (proven to be) equivalent in the underlying structure (or theory). Recall that $\vdash t \equiv u$ is a provable equivalence in the underlying structure. This rule is the tabular version of last chapter's *replacement* rule. In practice, the identities from the data type declarations suffice to prove most replacements, however more ingenious substitutions could be established with external rewriting tools or proof assistants.

Decision introduction

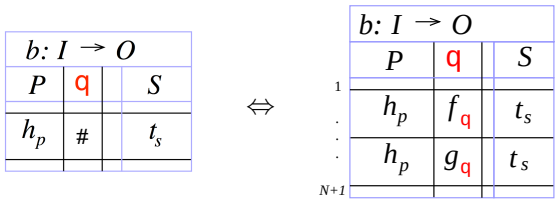


(58)

Decision introduction adds another test to the decision table. The guard column consists of entirely #s. By itself this introduction does not change architecture, however subsequent instantiation of guard terms (below) transforms selection terms in the action table into explicit enumerations in the decision table. The guard term q must have finite type, i.e. enumerated or bit-vector, so that subsequent expansions can exhaustively list its values in the decision table. Moreover, arbitrary use of variables in q can introduce combinational feedback; thus, the decision introduction

rule constrains q to result in a well formed system. In practice, banning the use of combinational variables in guard headings prevents feedback, though this rule is more restrictive than necessary.

Decision instantiation

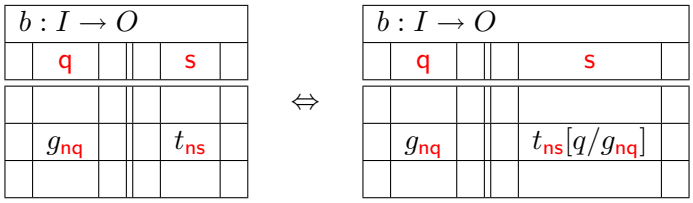


(59)

Having introduced a new test to a behavior table, instantiation is used to do case splitting. In the subject column, the expansive direction replaces # with a row for each possible value of q. This case enumeration duplicates the other guards h_p and actions t_s for the remaining columns. Decision identification (below) permits the subsequent propogation of test values to actions for term reduction.

The other direction ($\Leftarrow$) eliminates an unnecessary test by collapsing uniform actions and guards over a subject test q. Repeated application of this rule prepares a test for elimination from the decision table.

Decision identification



(60)

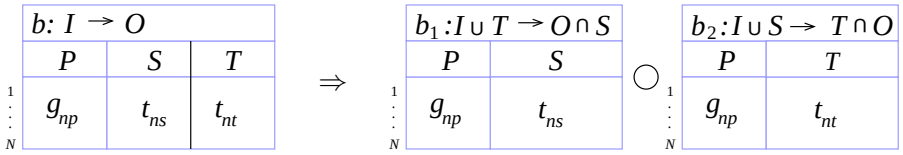
Decision identification substitutes the case value g_{nq} of a test q for instances of

the test term in the row's actions (and vice versa). This is analogous to replacing instances of a selector key in a selector branch with the corresponding branch value: e.g.,

$$sel(key, f(key), g(key)) = sel(key, f(0), g(1)) \tag{61}$$

when the key is a single bit.

Decomposition



(62)

Decomposition splits one table into two, both inheriting the same decision table. The *compose* operator connects the two tables to maintain the original dependence among the signals. Interpreting the tables as functions on streams—and reading '∪ and '∩ as list operations—$\mathcal{B}_1 \circ \mathcal{B}_2$ yields the system

$$\begin{array}{l} \mathcal{B}(I) \stackrel{\text{def}}{=} O \ \textit{where} \\ \quad (O \cap S) \ = \ \mathcal{B}_1(I \cup T) \\ \quad (O \cap T) \ = \ \mathcal{B}_2(I \cup S) \end{array} \tag{63}$$

Successive decompositions cause the connection hierarchy to deepen, often unnecessarily. Table editors should provide methods to flatten these connective structures. Let $\mathcal{C}$ denote a connecting block instead of a behavior table. The following equation expresses the *flatten* operation:

$$
\begin{array}{l}
\mathcal{B}(I) \stackrel{\text{def}}{=} O \ \textit{where} \\
\quad \vdots \\
\quad X \ = \ \mathcal{C}(Y) \\
\qquad \textit{where} \\
\qquad\quad \mathcal{C}(Y) \stackrel{\text{def}}{=} X \ \textit{where} \\
\qquad\qquad\quad X_0 \ = \ \mathcal{C}_0(Y_0) \\
\qquad\qquad\qquad\ \vdots \\
\qquad\qquad\quad X_n \ = \ \mathcal{C}_n(Y_n) \\
\quad \vdots
\end{array}
\quad \Rightarrow \quad
\begin{array}{l}
\mathcal{B}(I) \stackrel{\text{def}}{=} O \ \textit{where} \\
\quad \vdots \\
\quad X_0 \ = \ \mathcal{C}_0(Y_0) \\
\quad \vdots \\
\quad X_n \ = \ \mathcal{C}_n(Y_n) \\
\quad \vdots
\end{array}
\tag{64}
$$

The X_i are distinct from the $\mathcal{B}$'s internal variables. X is the tuple of $\mathcal{C}$'s signal visible externally; after flattening, X becomes superfluous since all of $\mathcal{C}$'s internal signals are in scope of $\mathcal{B}$'s connecting equations. Similarly, suppose that Y represents the set of signals in $\mathcal{B}$ that is exported to $\mathcal{C}$. After flattening, Y is superfluous since all of $\mathcal{B}$'s internal signals are in scope of $\mathcal{C}$'s lifted equations. Assume that formal placeholder variables represented by X and Y have the same name as the imported/exported signal names so that no additional α-conversion is necessary

Useless signal elimination

Input and output signals may be added to behavior tables without concern so long as the inputs and outputs of the encapsulating system remain the same. Simply adding inputs or outputs to a table does not introduce combinational feedback. This condition requires references to the new signals in the decision or action table, however the instantiation rules explicitly disallow combinational feedback.

Conversely, an unused I/O signal qualifies for elimination. We can remove input $i \in I$ to a behavior table if no action or predicate contains i as a subterm. A behavior table output may be removed when it neither supplies an input to another behavior

table (as dictated by the functional interconnect expression) nor doubles as an output of the encapsulating system.

Subterm Instantiation

$b : I \to O$		
P		s
g_{np}		t_{ns}

well formed $\Rightarrow$

$b : I \to O$		
P		s
g_{np}		$t_{ns}[u/\#]$

Signal Introduction

$b : I \to O$	
P	S
g_{np}	t_{ns}

y *fresh*
well formed $\Rightarrow$

$b : I \to O$		
P	S	y
g_{np}	t_{ns}	$\#_{ny}$

Sequential Identification

$b : I \to O$			
P	$S : t_n^{ini}$	$C : comb$	y : comb
g_{np}	t_{ns}	u_{nc}	r_{ny}

r *has no subterms in* I *or* C $\Updownarrow$

$b : I \to O$			
P	$S : t_n^{ini}$	$C : comb$	y : r$[s/t_s^{ini}]$
g_{np}	t_{ns}	u_{ns}	r$[s/t_{ns}]_{ny}$

Action Grouping

$b : I \to O$		
P		G
g_{np}		t_{ng}

$g \in G$ *uniformly sequential or combinational* $\Leftrightarrow$

$b : I \to O$			
P		$G : comb$	g
g_{np}		n^{th}(g)	$[t_g]_n$

Signal Elimination

$b : I \to O$		
P	S	R
g_{np}	t_{ns}	u_{nr}

$r \in R$ *unused* $\Rightarrow$

$b : I \to O$	
P	S
g_{np}	t_{ns}

Combinational Identification

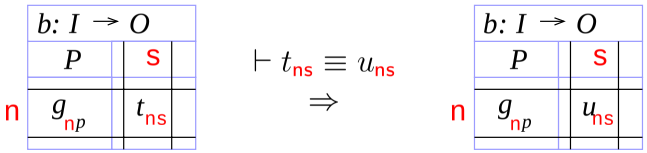
$b : I \to O$			
P		s	y : comb
g_{np}		t_{ns}	r_{ns}

well formed $\Updownarrow$

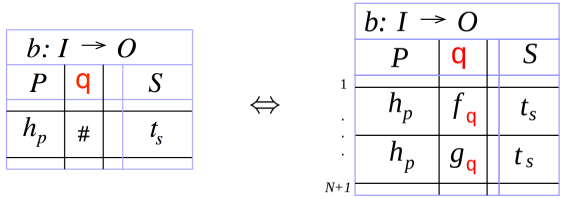
$b : I \to O$			
P		s	y : comb
g_{np}		$t_{ns}[r_{ns}/y]$	r_{ns}

Figure 4: Transformation summary

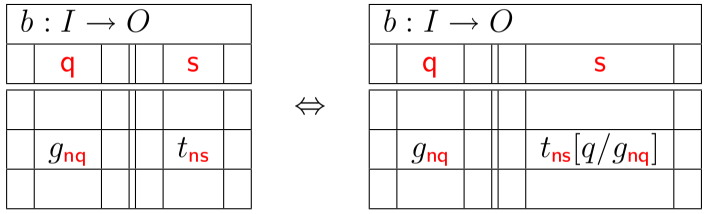
Replacement



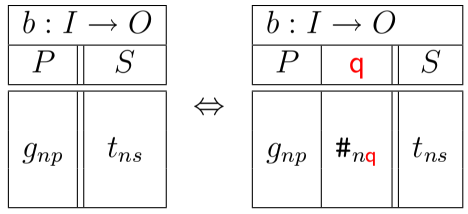
Decision Instantiation



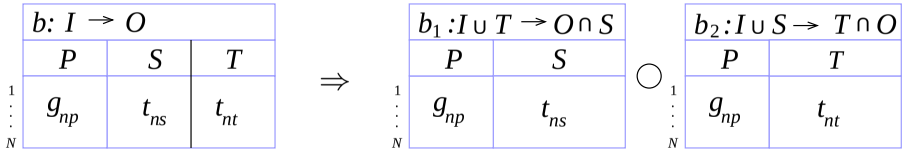
Decision Identification



Decision Introduction



Decomposition



connected by

$$\mathcal{B}(I) \stackrel{\text{def}}{=} O \text{ where}$$
$$(O \cap S) = \mathcal{B}_1(I \cup T)$$
$$(O \cap T) = \mathcal{B}_2(I \cup S)$$

Useless I/O Signal Elimination

Flatten Interconnect

Figure 5: Transformation summary *(cont.)*

Chapter 5

Starfish

Starfish is implemented as two communicating processes. The first process maintains, transforms, and analyzes the behavior table data structures. The second process is the user interface for table display and script execution. The two processes communicate through Unix FIFOs using XML to encode the tables and commands destined for the GUI, and the starfish command language to communicate transformations and session commands to the computational back-end.

5.1 Computational Engine

The computational backend hosts the primary representation, analysis, and transformation of behavior table systems. Like DDD, this transformation package is written in Scheme. However, Starfish represents systems directly as behavior tables, rather than systems of stream equations. The major components of its computational back-end have significant interdependencies, but are roughly classified into six categories of functionality: table representation, type system, component addressing, transformations, property analysis, and command interpreter.

5.1.1 Language prerequisites

The code for the system makes extensive use of specially designed Scheme macros. The first macro package, `define-dt`, extends the inductive data type schema developed by Friedman [27]. It lets programmers define sets of related data type variants, each possessing multiple fields. The fields are subject to arbitrary programmer-defined predicates, including the property of belonging to the general subject data type or a specific variant. This paradigm is well suited for representing language expressions; for example, the general data type, *term*, has variants *constant*, *variable*, *function application*, etc. Function applications have two fields: *function* which satisfies the predicate `function?`, and *args* must satisfy the compound predicate `(list-of term?)`, ensuring that the arguments belong to the inductive data type, *term*. All field properties are verified at construction time. Starfish's implementation of this facility departs from the Friedman implementation by introducing predicate and accessor bindings for each variant of the data type only into the scope where it is declared.

Nearly every internal data structure in Starfish is represented with `define-dt`. For the most part, Starfish maintains a functional programming style in which data structures are not modified after initialization. Results of table transformations are always newly constructed tables rather than small mutations of the input system. Since the data structures are deeply nested, and the programmer needs access to large numbers of fields for analyzing transformations and constructing the new system, Starfish distinguishes and binds arbitrary sets of data-tree nodes with a special pattern matcher, `match-dt`. A polymorphic constructor `change-dt` leverages the same branch specification method to define modifications to existing trees as sets of

branch substitutions.

Example 5.1: Starfish internally represents terms with the following data structure:

```
(define-dt term
  ([type type?])
  ((const ([name symbol?]))
   (var ([name symbol?]))
   (uspec ())
   (app ([func func?] [exps (list-of term?)]))
   (sel ([key term?] [exps (list-of term?)]))
   (tuple ([exps (list-of term?)]))
   (proj ([index integer?] [exp term?]))
```

The data type's name is `term` and every variant has the field `type`, which must satisfy the predicate `type?`—a predicate generated by a similar data type declaration for `type`. The variant `const` has the mandatory `type` field and a symbolic `name`. The `sel` variant adds `key` and `exps`, which are a term and list of terms respectively.

One well formedness check assures that they `key` term is a finite type. Given a `sel` term one could apply a sequence of field accessors to assess the type. For example:

```
(finite-type? (term->type (sel->key t)))
```

This becomes tedious, particularly when a term has multiple subfields of interest. The `match-dt` facility allows one to bind arbitrary subfields (including parent and child nodes) of a specified pattern. In recognition that the name of a field is not always a convenient binding name in a particular scope, the `match-dt`-form specifies the pattern and binding variables with different expressions:

```
;; match-dt approximate form:
;; (match-dt <term>
;;  ([<pattern> <binding-list>] <exp> ...) ...)

(match-dt t
 ([(sel (key :: const (type :: enum-type alg)  name) exps)
   (    (key ::       (ktype ::         kalg) cname) exps)]
   (printf "The selector key's enumerated algebra is ~s" alg))
 ([(sel (key :: const type name)    exps)
   (  (mykey ::      ktype cname) myexps)]
   (printf "This selector's key is a constant ~s" cname))
 ([(sel key exps) (mykey myexps)]
   (printf "This selector's key is ~s" (term->text mykey))))
```

The branches of this `match-dt` expression are in decreasing order of complexity since they match subsets of each other. The last clause matches `t` if it is a `sel` variant of `term`. In case of a match, it binds the key and expressions to `mykey` and `myexps` and evaluates `printf` expression in the context of these bindings. The second branch further specifies that the selector's key must also be a `const`. The binding list assigns the key to `mykey`, the key's type to `ktype`, the constant name to `cname` and the selector expressions to `myexps`. This specification of subfields and subsequent binding my apply arbitrarily deeply in the tree structure. The first clause further specifies that the type is an enumerated type and binds the corresponding enumerated algebra to the variable `alg`. There is no requirement to specify all fields of a particular variant, only the ones of interest.

The `change-dt` constructor specifies branches in the same way, substituting new expressions for the specified branches. The functional way to change the type of a selector term, `t`, accesses all of reused the fields and applies the basic constructor to make the new term with the newly specified type:

```
(make-sel new-type (sel->key t) (sel->exps t))
```

The change-dt accomplishes the equivalent action with

```
(change-dt (sel type) (new-type))
```

As with `match-dt`, only the fields of interest need specification and there may be multiple substitutions. Substitution specifiers may not include fields where one is a direct descendant of another. The following statement changes both the type and the constant name of a matching selector:

```
(change-dt (sel type (key :: const name)) (new-type (new-cname)))
```

Another ubiquitous but non-standard form in the Starfish code base is `poly-lambda`. This form allows programmers to "overload" a function to accept multiple input signatures. The following code schema are equivalent:

```
(poly-lambda
 [([arg0 pred0] ... [argn predn]) exp exp1 ...] ...)

(lambda args
  (cond
   [(and (eq? (length args) n*)
 (let ([arg0 (list-ref args 0)]
       ...
       [argn (list-ref args (- n 1))])
   (and (pred0 arg0) ... (predn argn))))
    (let ([arg0 (list-ref args 0)]
  ...
  [argn (list-ref args (- n 1))])
      exp exp1 ...)] ...))
```

Example 5.2: Define `f` as follows:

```
(define f
  (poly-lambda
   [((x integer?) (y symbol?)) (list x y)]
   [((x integer?) (z integer?)) (+ x z)]
   [() 0]
   [() 2]
   [fail f])) ;; reports "Error in f: ..."
```

Then we have the following behavior for `f` in an interactive session:

```
> (f)
0
> (f 1 2)
3
> (f 1 'a)
(1 a)
> (f 1)
Error in f: incorrect argument structure (1).
Type (debug) to enter the debugger.
```

The `poly-lambda` form was particularly useful in adapting behavior table transformations to new methods of specifying system components. As the argument specification methods changed, the original transformation and analysis could be reused by adding another argument clause that translated the new specification method into the old one and recursively called the original transformation.

5.1.2 Type system

Starfish's type system is integrated with its representation of terms. As depicted in Example 5.1, a term's type is the only mandatory field for every term variant. The type system's declarative structures, introduced in Section 3.2.2, define the symbol table for terms. A global object, `alg-reg`, stores function and constant symbols along with their IO-signature or type and structure of declaration (e.g., `push` is part of the

`stack` declaration).

The symbol table is the foundation for parsing and type checking term expressions. Term syntax indicates function application with parenthetical s-expressions, tuples with square brackets, unspecified terms with `#`, and projectors with ordinals (e.g., `1st`, `2nd`, `3rd`). Valid constants and functions are specified in the symbol table, and variables definitions come from the environment of the behavior table (i.e., the signal names).

Ideally, a term string would have a unique type and the system could automatically determine it. In this case, the user would never need to specify the type of a subterm because the system would *infer* it from the types and signatures specifications in the symbol table and environment. Since terms in this system are first order, the inference problem is greatly simplified. However, parameterized types, integer-parameterized bit vectors, polymorphic tuples, and unspecified terms introduce some ambiguity to the system; for instance, assuming that stacks are parameterized over a content type, what is the type of `empty-stack`? Worse yet, what is the input type requirement for the projector `3rd`?

Starfish takes a pragmatic approach to the inference problem. First, Starfish inferences types with an algorithm that works "well enough." This thesis makes no claims to its completeness. The algorithm proceeds by recursive descent through a term's data structure. It begins with an estimate (possibly trivial) of the type and branches according to subject's term variant. In addition to the fully-specified types arising from the augmented multisorted algebra type system presented in Section 3.2.2, Starfish's type-inferencing algorithm encodes type requirement information

with a number of partly defined types: `unknown-type` makes no requirements on the type, `unknown-bvec-type` indicates a bit-vector of unknown size, `unknown-key-type` indicates that the type is finite and has a certain size, `unknown-tuple-type` is a recursive variant that indicates that the type is a tuple (of various unknown types) and places a lower bound or exact specification on its length, `unknown-param-type` is a recursive variant that indicates that the type belongs to a certain parameterized algebra (e.g., stacks), but that its parameters are not fully specified.

The algorithm accepts a partially typed subject term and a type requirement expression (either an unknown type as above, or a fully specified type), and returns a subject term where every subterm's type field is a fully specified type. The algorithm presumes that all types specified within the term's structure are correct. When it encounters a constant, it looks up the constant's declaration algebra and tries to *merge* or *unify* the type requirements with the declaration. If the requirements are incompatible, the algorithm throws a *type mismatch error*, noting the difference between requirements and declaration. If they are compatible but do not fully specify the type, the algorithm throws a *type ambiguity error* (unless requested not to). Otherwise, a type is fully specified and satisfies the requirements; the type inferencer returns the labeled constant. A similar process works for variables. For recursive term variants, the type inferencer is run on each of the subterms. The results are merged according to the variant's properties. For instance, a selector must have uniform types in its branches. In this case, recursive calls on the branches do not fail when their result is an ambiguous type. Instead, ambiguous types are accepted and unified across branches.

The most complex case occurs when inferring the type of a function application that returns a parameterized type. The type inferencer tries to evaluate the "simplest" input parameters first. It classifies input parameter type complexity according to the function's input signature: *non-schema* inputs have fully defined types, *flat-schema* input types are type variables, and *deep-schema* inputs have type structures with type variable subtypes. The inferencer evaluates these three categories of subterms in order, and merges the results into a type variable environment as it completes inferences on each input argument. After inferring type values for the argument subterms, the inferencer estimates the output type with variables from the type environment, and the unifies estimates with the incoming requirements; mismatch and ambiguity are handled as before.

Certainly there are cases where a term's type is logically deducible, but exceed the algorithm's inferencing capability. For this reason, Starfish supports type annotation in its term expressions (see Figure 2 from Section 3.2.2). In practice, this leads to very few annotations. Even in the larger case studies, terms tend to have simple structures and simple types. Annotation has been limited to identity specification in data-refinement declarations.

5.1.3 Component addressing

Specifying transformations is challenging because the primary operand, the system specification, is a deep tree structure with potentially duplicate branches. Thus specifying a subcomponent by value in many cases does not uniquely determine the target of the transformation. Furthermore, this approach can be cumbersome when target

components are complex—e.g., an entire behavior table. To mitigate this problem, an addressing system specifies each component by a sequence of branch numbers that enumerate the component's position in the data tree.

Generally, the address can be split into three : a subsystem node address, a node component (such as a table column, cell, or output) and subterm node address. A subsystem node address specifies a behavior table or connecting expression with lists of non-negative integers that reference which child or sub-branch the path follows. In command specification system paths are either specified with lists (e.g., `'(0 2 1 1)`) or with the system path constructor `(make-sys-path 0 2 1 1)`; the implementation uses both methods, reflecting evolving design decisions throughout its development.

Within a table or connecting expression, a data type variant specifies the subcomponent in question and carries further specification data. For instance, there is a variant for specifying a signal (column in the action table) in a behavior table, `sig-addr`. It has two fields: `sys` a subsystem address which specifies the behavior table in the behavior table hierarchy, and a non-negative integer specifying the action table's column. Another variant, `act-subterm-addr` specifies a subterm in a behavior table's action table. It has three fields: a `sig-addr` to identify the column, a row number, and a subterm address.

Like system node addresses, subterm addresses are essentially a list of numbers specifying a path to a node in the subterm tree. For example, the subterm address $(0, 1, 0)$ references the subterm $f(1, c)$ in the term $[sel(k, f(1, c), 2), g(x), 0]$ (the top-level term is a 3-tuple), while the path (2) references the subterm 0, and the empty path references the top-level 3-tuple.

This numerical addressing scheme gives a unique name to the components of a behavior table hierarchy, however, using the full address for every transformation parameter results in clutter and redundancy. Most transformations have a high degree of parameter locality and only reference components within the same node. Furthermore, the subclass of node components are usually determined by context of the transformation. For instance, a signal introduction or combinational identification necessarily operate on the action table. Thus most transformation commands take a sub-system path as their first argument and a collection of local references for the following arguments—the system tree is part of the global state, and implicitly an argument to each transformation.

Early versions of Starfish reference table cells with row and column numbers, while subterms were further specified by a subterm path. Although this method was sufficient for specifying transformations, it proved tedious to update when small changes were made to derivations scripts—i.e., sequences of transformations. Many transformations change the table dimensions and thus change the row and column numbers of transformation parameters. Later versions of the many transformation commands allow row and column specification by signal name (column heading in the action table) and guard (the sequence of constants in the decision table). This simple change greatly increased resilience to small changes in derivation scripts.

The numerical absolute addresses, though poorly matched for specifying scripts, are returned by Starfish's property matcher. The property matcher accepts a subnode address and a term predicate (as a Scheme expression), and returns a list of numerical addresses pointing to terms matching the property; for example, designers may want

to know the locations of all terms belonging a certain type. These addresses are too crude for direct human evaluation, however Starfish uses them to specify colorization points to the display interface.

5.1.4 Analysis

Starfish bundles most of its analytical activities with transformations. A transformation performs roughly three levels of analysis: parameter well-formedness, satisfaction of transformation requirements, and heuristic analysis to produce the most mature transformation result. Starfish has limited support for user directed analyses through its context-free property matcher that returns references to all subterms of a specified action table satisfying a specified predicate. The Starfish front-end colorizes lists of action subterm addresses, and the resulting display is a decision making aid for the designer.

Since the designer specifies transformations at the command line, they must guard against incorrectly specified references. For example, the first argument to all transformations is sub-system path that points to the node of interest. Every transformation checks that the path points to a node in the current state, since it is possible to specify an undefined path—e.g., specify a child node in a flat system. Similarly, transformations check signal name specifications, guard specifications, subterm paths, and other references to assure that they map to actual system objects.

The next level of analysis confirms whether the specified system components meet the transformation prerequisites. This analysis checks the conditions at the level presented in table algebra Section 4.2: some transformations only work on combinational

signals, others require the actions to have a certain form, etc. The conditions are generally straightforward—e.g., confirming that a signal is completely unspecified—and are verified by the transformation function prior to the system update. Since these properties simply validate transformation inputs, there has been little effort to separate this pre-analysis from the transformation itself.

The most common non-trivial analysis is signal dependence: signal `a` *immediately depends* upon signal `b` when an action subterm of `a` contains a variable reference to `b`. *Signal dependence* is the transitive closure of immediate dependence. Signal dependence is the foundation for combinational cycle detection, external signal elimination, and identifying groups of unused internal signals.

Some transformations perform heuristics to automate transformation sequences and decisions that could theoretically be left to the designer. For instance, decomposition and hierarchy flattening result in signal sharing among components. A gross transformation could simply make all signals available to all components without regard for use. Starfish only provides components with the input signals necessary to close the table expressions. The reduction requires a signal dependency analysis over the subcomponents of the decomposition. The higher level system factorization transformations (presented in Chapter 6), reduce the number of external component operations with parameter ordering heuristics. It does not produce optimal results in general but rather a plausible beginning for manual refinement.

Starfish implements an action subterm predicate matcher. It accepts a table reference and a term predicate (written in Scheme), applies the predicate to each *subterm* of the action table, and returns a collection of subterm addresses that satisfy

the predicate. The predicates are context-free in the sense that they can only evaluate the data structure of the term itself; they are unaware of the term's position in the table. Common uses for this are to identify subterm types in the table or to find instances of certain function applications. Once identified, the Starfish user interface can color disjoint lists of subterms to display the results. These queries aid factorization decisions where the goal is to encapsulate function applications or abstract signals with architectural components.

5.1.5 Transformations

Starfish implements the core table transformations presented in Section 4.2, term identities over the special term variants (i.e., selectors, tuples, and projectors), and higher level transformations that automate more complex goals such as system factorization (Chapter 6), serialization (Chapter 7) and data refinement (Chapter 8). The details of these last transformations are covered in their respective chapters.

Once the type system and addressing scheme were established, the core transformations became straightforward to implement. The following is a quick summary of the core forms and special term identities. Angle brackets enclose descriptive parameter names and required type—e.g., <*name* : *type*>. While almost all transformations are polymorphic and accept a variety of input signatures, description is limited to one input signature—usually the most general.

Subterm Instantiation

```
(specialize-term <table:sys-path> <sig-name:string>
          <guard:list-of-strings> <new-term:string> <subterm:list-of-ints>)
```

confirms that `table` references a table in the system tree, that `subterm` references a `#` in the action term determined by the row, `guards`, and column, `sig-name`, that `new-term` is a well defined in the table scope, that instantiation of `#` by `new-term` does not introduce combinational feedback, and finally commits the instantiation to the system description.

Signal Introduction

```
(add-act-col <table:sys-path> <sig-name:symbol> <type:string> <src:string>)
```

confirms that `table` references a table in the system tree, `sig-name` is fresh in the table scope, `type` is a declared type, `src` is `comb` or `seq`, and then augments the specified table with a new signal of specified type and source (i.e., combinational or sequential) where all action terms are `#`.

Signal Elimination

```
(remove-act-col <table:sys-path> <signals:list-of-symbols>)
```

confirms that `table` references a table in the system tree, `signals` is a list of signal names in the table scope, no other terms are dependent upon the presence of the specified `signals`, and then removes these signals from the action table of the specified table.

Combinational Identification

```
(apply-comb-ident <table:sys-path> <guard:list-of-strings>
                  <sig-name:string> <subterm:list-of-ints> <comb-sig:string>)
```

confirms that `table` references a table in the system tree and that `subterm` is a valid path is the action term determined by the row, `guard`, and column, `sig-name`. If `subterm` references a variable instance of the signal `comb-sig`, then it substitutes

`comb-sig`'s action term for the variable instance. If `subterm` references `comb-sig`'s action term, then the transformation substitutes a variable reference to `comb-sig`'s for the action term.

Sequential Identification

```
(unroll-comb <table:sys-path> <sig-name:symbol>)
```

confirms that `table` references a table in the system tree, that `sig-name` is a combinational signal of uniform action over the column, and that the action term only contains variable references to sequential signals. The result turns `sig-name` into a sequential signal and replaces sequential variables in the signal's actions by the defining expressions (p. 66). Although the transformation is valid in both directions, Starfish does not implement the inverse direction which transforms sequential signals into combinational.

```
(eliminate-comb-refs <table:sys-path> <sig-indices:list-of-ints>)
```

is a helper that recursively replaces combinational variables with their defining expressions. Since `unroll-comb` requires exclusively sequential variable references in the target's update actions, designers can prepare the target signal by applying `eliminate-comb-refs`. The transformation confirms that `table` references a table in the system tree and that `sig-indices` reference a set of sequential signals before expanding the actions.

Action Grouping

```
(group-act-cols <table:sys-path> <new-sig:symbol>
                <sig-names:list-of-symbols>)
```

confirms that `table` references a table in the system tree, that `sig-names` are uniformly sequential or uniformly combinational, and that `new-sig` is an unused name in

the table scope. The transformation adds a new signal, `new-sig`, to the table whose type is the order tuple of signal types specified by `sig-names`, and whose source (combinational or sequential) aligns with that of `sig-names`. The new actions are the tupled actions of `sig-names`. The signals specified in `sig-names` are retained as combinational projections of `new-sig`, so that prior references to these signals are still valid.

```
(split-act-col <table:sys-path> <sig-name:string>
               <new-sig-names:list-of-symbols>)
```

confirms that `table` references a table in the system tree, that `sig-name` is a tuple type, that `new-sig-names` are fresh in the table scope, and that the number of `new-sig-names` is the same as the number of tuple components in `sig-name`. The transformation creates new signals from each projected component of `sig-name`; they have the projected type and share the `sig-name`'s source. Projections of tuple constructors are simplified in the resulting action terms. The table retains `sig-name` as a combinational signal defined as a tuple over `new-sig-names`, so that prior references to `sig-name` are still valid.

Replacement

```
(apply-alg-ident <table:sys-path> <guard:list-of-strings>
                 <sig-name:string> <subterm:list-of-ints> <id:symbol>)
```

confirms that `table` references a table in the system tree, that `subterm` is a valid path in the action term determined by the row, `guard`, and column, `sig-name`, and that `id` is a declared identity in the type database. The transformation logic automatically determines which direction to apply the equivalence by pattern matching. In some unusual cases, both directions may match the subterm; the default transformation

applies the left-to-right rewrite (as specified in the declaration). Another form of the transformation forces the right-to-left rewrite.

Decision Introduction

```
(add-pred-col <table:sys-path> <test:string>)
```

confirms that `table` references a table in the system tree, that `test` is a well formed term in the table scope of finite type, and that no variable references to combinational signals are subterms of `test`—this constraint eliminates one potential source of combinational feedback. The transformation adds a new column of `#` to the decision table under the `test` heading.

```
(remove-pred-col <table:sys-path> <test-index:int>)
```

confirms that `table` references a table in the system tree, that `test-index` is inside the column bounds of the decision table, and that the referenced column does not specify any guards—i.e., that the column is uniformly `#`. If satisfied, the transformation removes the referenced column.

Decision Instantiation

```
(expand-row <table:sys-path> <guard:list-of-strings> <test:string>)
```

confirms that `table` references a table in the system tree, that `test` and `guard` are valid, that the term corresponding by `test` in `guard` is `#`. Then the transformation creates a guard for each constant of the finite type corresponding to `test`. The new table actions duplicate the one referenced by `guard`. As a convenience, selectors in the action row that branch over `test` are simplified according to the new guard constant.

```
(collapse-rows <table:sys-path> <test-index:int>
               <guard-indices:list-of-ints>)
```

confirms that `table` references a table in the system tree, that `test-index` and `guard-indices` are inside the column bounds of the decision table, that guards referenced by `guard-indices` are identical in all columns except the one referenced by `test-index`, and that the guards exhaust the constants for the finite type—e.g., if collapsing over a two-bit vector, the guard set must contain all four values (0b00, 0b01, 0b10, 0b11) in the `test` component. The resulting decision table replaces the guarded rows with a single guard that has a `#` in the `test` component and the same values for the other tests. The resulting action condenses the action set with a selector keyed by `test` branching over the values of each element of the action set as described in (59)

Decision Identification

```
(apply-pred-ident <table:sys-path> <guard:list-of-strings>
                  <sig-name:string> <subterm:list-of-ints> <test:string>)
```

confirms that `table` references a table in the system tree, that `test`, `guard`, `sig-name`, and `subterm` are valid, and that `subterm` matches the expression `test` or the `test` constant of `guard`. The transformation changes `subterm` into the `test` component of `guard` or back to `test`.

Decomposition

```
(split <table:sys-path> ((<tname:string> . <tsignals:list-of signals>) ...))
```

A partition of a table's signals and a name, `tname`, for each set of the partition, `tsignals`, constitutes decomposition specification. The transformation confirms that `table` references a table in the system tree, that the symbols reference signal names, that the partition sets disjointly cover the action table, and that the `tname`s are unique. The transformation replaces the table with a connecting node that minimally

closes—i.e., supplies exactly the input signals for all of the unbound variable references in a table—the specified subtables. The subtables have the same decision table as the target table but only the subset of internal signals as specified by `tsignals`.

`(flatten-sys-node <gr-node:sys-path>)`

confirms that `gr-node` references a node with at least one grandchild in the system description. It removes the parents nodes of the grand children, and establishes the minimal connections necessary to close the grandchild nodes in context of the grandparent signals.

Useless Signal Elimination

`(remove-input-signal <node:sys-path> <isig:string>)`

confirms that `node` is a valid address in the system description, and that `isig` is an unused signal in `node`. The transformation removes the input signal and, when applicable, reflects this change in the connecting equations of the parent node.

`(remove-output-signal <node:sys-path> <osig:string>)`

confirms that `node` is a valid address in the system description, that `osig` is an output signal in `node`, and that there are no references to `osig` in `node`'s parent.

"Named" Signal Manipulation

Starfish adds a convenience feature to behavior tables called a *named signal*. Combinational signals frequently have the same action in every row; for example, the signal might unconditionally project from a tuple (named signals perform this service in `group-act-cols`). Rather than clutter the table, the designer can express these combinational signals as unguarded equations (`<name>=<term>`), called *named signals*; they are semantically equivalent to combinational signals with uniform action.

Starfish introduces named signals directly with `add-named-signal`, and converts form with `name->signal` and `comb-signal->name`.

`(name->signal <table:sys-path> <name:symbol>)`

confirms that `table` references a table in the system tree and that `name` is a named signal in the table scope. It creates a combinational signal in the action table with name and type of `name` and with uniform action updates corresponding to the `name`'s defining term.

`(comb-signal->name <table:sys-path> <name:symbol>)`

confirms that `table` references a table in the system tree and that `name` is a combinational signal of uniform action in the table scope. It removes the combinational signal from the action table and creates a named signal defined by this unique action term.

`(add-named-signal <table:sys-path> <name:symbol> <term:string>)`

confirms that `table` references a table in the system tree, that `name` is fresh identifier in the table scope, that `term` is well defined in the table scope, and that `term` does not introduce any combinational feedback. Once confirmed, it adds the proposed signal to the table. This transformation is equivalent to introducing a new combinational signal, instantiating all actions to the same term, and then converting to a named signal.

Special Term Identities

The special terms variants (i.e., selectors, tuples, projectors, and uspecs) obey identities that can not be expressed in Starfish's type declaration facility. Starfish implements these directly. The first two parameters to each of these are `<table:sys-path>`

and `<subterm:subterm-addr>`, which determine the subject action. The identities confirm that these are valid references, and then perform the specific pattern match as described below.

```
(eliminate-sel <table:sys-path> <subterm:subterm-addr>)
(expand-to-sel <table:sys-path> <subterm:subterm-addr> <c:string>)
```

These two transformation respectively implement the left-to-right and right-to-left forms of the following identity:

$$sel(c, t_{c_1}, \ldots, t_c, \ldots) = t_c \tag{65}$$

`eliminate-sel` confirms that `subterm` references a selector with constant key, `c`, and then replaces the term with the `c`-branch of the selector. After confirming that `c` is a constant of finite type, `expand-to-sel` replaces the subject term with a selector branching over `c`. The `key` branch contains the original term, while the other branches are consistently typed `#` terms.

```
(eliminate-proj-tuple <table:sys-path> <subterm:subterm-addr>)
(expand-to-proj-tuple <table:sys-path> <subterm:subterm-addr>
                      <n:int> <type:string>)
```

These two transformations respectively implement the left-to-right and right-to-left forms of the following identity:

$$n^{th}([\ldots, t_n, \ldots]) = t_n \tag{66}$$

`eliminate-proj-tuple` confirms that `subterm` references a projector over a tuple constructor and then replaces it with the projection term. In the opposite direction, `expand-to-proj-tuple` additionally specifies the projector with `n` and the tuple type

with `type`. The `type` must have at least `n` components. The result is a projection of a tuple whose n^{th} component is the original term and the remaining components are correctly typed `#` terms.

`(tuple-of-uspec-is-uspec <table:sys-path> <subterm:subterm-addr>)`

This transformation implements the following identity:

$$[\# : type_1, ldots, \# : type_n] \Rightarrow \# : [type_1, ldots, type_n] \tag{67}$$

It replaces a tuple of `#`-terms with a `#`-term of type tuple.

`(proj-of-uspec-is-uspec <table:sys-path> <subterm:subterm-addr>)`

This transformation implements the following identity:

$$n^{th}(\#) : [\ldots, type_n, \ldots] \Rightarrow \# : type_n \tag{68}$$

It replaces a projection of a `#`-term with a `#`-term of the correct component type.

5.1.6 Command Interpreter

Starfish's command interpreter provides a rudimentary command line interface for loading system specifications, executing transformations, and derivation script management. The command interpreter follows the standard *read-eval-print* pattern. It is an editor for the system state which is an implicit argument to every command. The interpreter accepts s-expression style commands through a redirectable input port. The command interpreter evaluates three classes of commands: file commands, transformation commands, and history commands. After command evaluation, the

interpreter calls a display routine, implemented as a thunk, that converts the resulting tool state into the XML display language accepted by the graphical user interface. This string is sent over the display output port, implemented as a Unix FIFO, or *named pipe*.

The command interpreter maintains three kinds of global state: the type registry, the design state, and the command history. The `load-spec` command reads a file containing type declarations followed by the system specification. It first populates the type registry and then does a well-formedness check on the system specification. Once validated and parsed, the command interpreter sets the specification data structure as the initial state of the history stack and sets the *redo* command stack to empty. A derivation script is a `load-spec` command followed by a sequence of transformation commands. The `load-session-script` command opens derivation scripts in two steps: first it loads the system specification with `load-spec`, then it populates the *redo* command stack with the derivation's transformation commands. The `save-session-script` command saves derivation scripts by copying the initial `load-spec` command and subsequent transformation commands from the history into a specified file.

Transformation commands operate on the system state at the top of the history stack. They apply the specified transformation, and push new a frame onto the history stack. The history stack frame contains the transformation result, the transformation command that produced the result from the previous state and the display thunk. History commands let the user undo transformations by popping the history stack frames. As frames are popped, Starfish pushes the commands (but not the remainder

of the stack frame) onto the redo-stack. Redo commands re-execute transformations on the redo-stack; when executing multiple transformations serially, Starfish sends only the last command's output to the display interface.

5.2 Display

The display process renders the behavior table hierarchy, serialization tables, and the current derivation script with the Java Swing API. Figure 6 shows the features of Starfish's behavior table rendering. The window is vertically split with a hierarchy widget on the left hand side and the node contents on the right. The externally visible input and output signals occupy the right-hand-side's heading. When the node connects subsystems, the body of the right hand side shows the defining equations and internal signals. When the node is a leaf, the body of the right hand side is a behavior table.

Figure 7 illustrates the mapping between the connective nodes and an architectural block diagram. The parent block, `Shift and Add Multiplier`, connects two subcomponents. It has three inputs that are internally labeled by `go`, `a`, and `b`. The first subsystem, `Shift and Add Multiplier 1`, produces three internal signals: `inst`, `done` and `acc`. The last two, `done` and `acc`, are made visible externally. Similarly, `Shift and Add Multiplier 2` produces the internally visible signals `u` and `v`. Internal input and output names for each subsystem are indicated in the block diagram, however Starfish's hierarchical rendering only indicates these names in the behavior table subnodes (Figure 6).

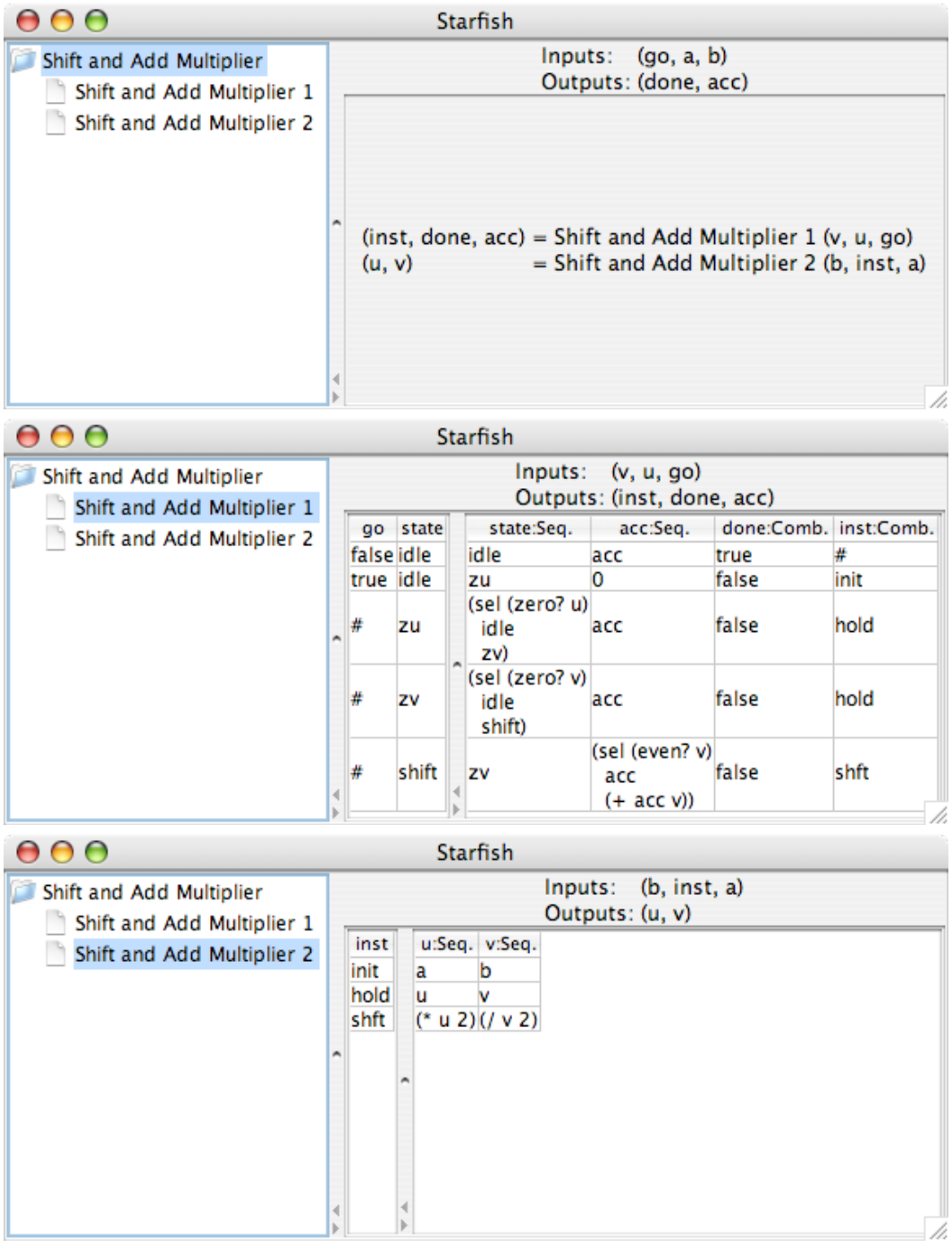


Figure 6: Starfish renders a system of behavior tables using the tree widget on the left side to navigate the connection hierarchy while displaying node contents on the right.

To save space, screenshots are generally limited to the behavior table hierarchy's most relevant components. Examples will usually omit the hierarchy tree on the right, and sometimes the input and output signals. In large specifications, the number of columns and rows can exceed the canvas size. A scroll bar provides access in the tool. Since most transformations have a localized effect and do not sparsely change the behavior table over a wide area, the exposition can illustrate transformations by restricting the rendering to a few key cells.

5.3 Interprocess communication

Interprocess communication consists of the transformation engine's command interpreter sending XML encoded commands to the display process through a FIFO, and a display widget sending s-expression commands to the transformation engine command interpreter through another FIFO.

The display process interpreter accepts the following commands: DisplaySys, Select, SetProp, DisplaySer, CommitSer, DisplayCmds, and Batch. These commands are encoded as XML documents. The display process listens on the command input FIFO and parses incoming commands using the low-level SAX XML parser [67]. The parser returns only after reaching the *end-of-file* character, so this communication is synchronized by closing the input FIFO. The most common display command is `DisplaySys`, which transmits the entire behavior table hierarchy as its argument. The command causes the display to overwrite its existing representation of the system. Starfish issues a `DisplaySys` command whenever the behavior table state changes.

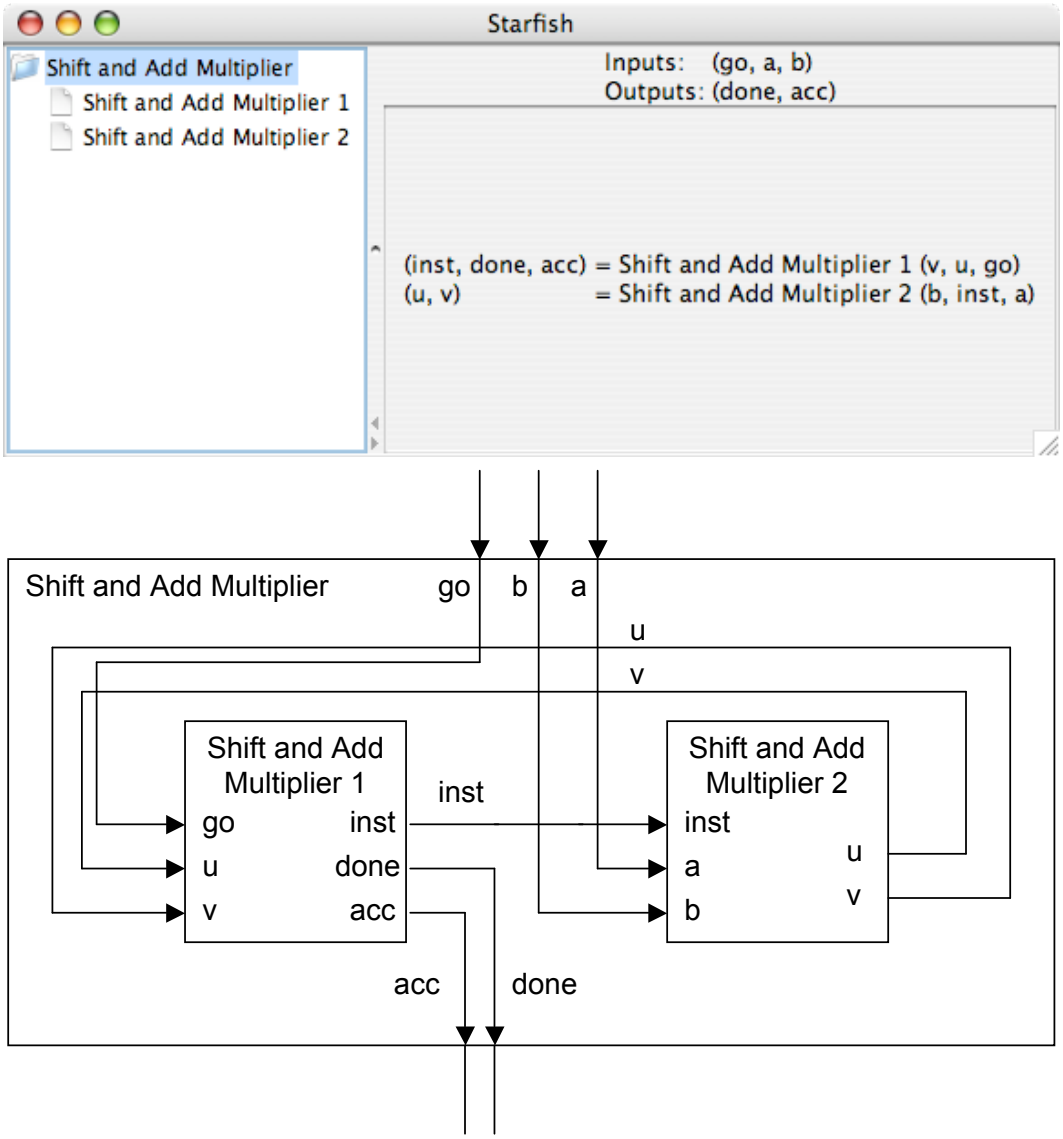


Figure 7: The connective equations convey the same information as this architectural block diagram. Each line is a labeled signal whose scope is determined by the enclosing block. For instance, the `go` signal in the internal node, `Shift and Add Multiplier`, is only visible inside the connective block; it is *not* visible inside of the leaf node `Shift and Add Multiplier 1`, although this particular specification has used the same name its reference to the incoming signal. Similarly, `a`, `b`, `u`, and `v` need not maintain their names as they cross hierarchical blocks.

`Select` accepts a numerical system address as an argument; the display process highlights the referenced term in its current hierarchy with the color red. `SetProp` is similar, but parameterizes the colorization. `DisplaySer` sends the contents of a serialization table (defined in Chapter 7), but not the behavior table hierarchy, to the display process. `CommitSer` eliminates the serialization table display. `Batch` packages multiple commands together. `DisplayCmd` sends the current derivation script to the display process; it displays transformations insides a list widget, allowing the user to skip forward and backward in the derivation.

The display process sends only limited commands to the transformation process. The derivation script list widget sends `(undo i)` and `(redo i)` requests to the transformation back-end's command interpreter. In practice, the derivation scripts are assembled by hand in a text editor and tested using this display widget to step through each transformation.

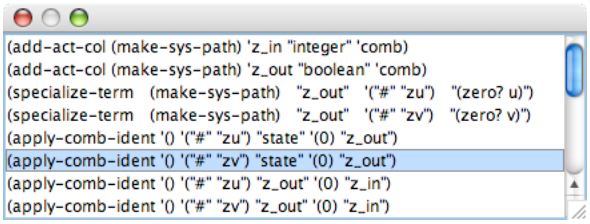


Figure 8: The script navigator lets the designer jump to an arbitrary stage of the derivation script.

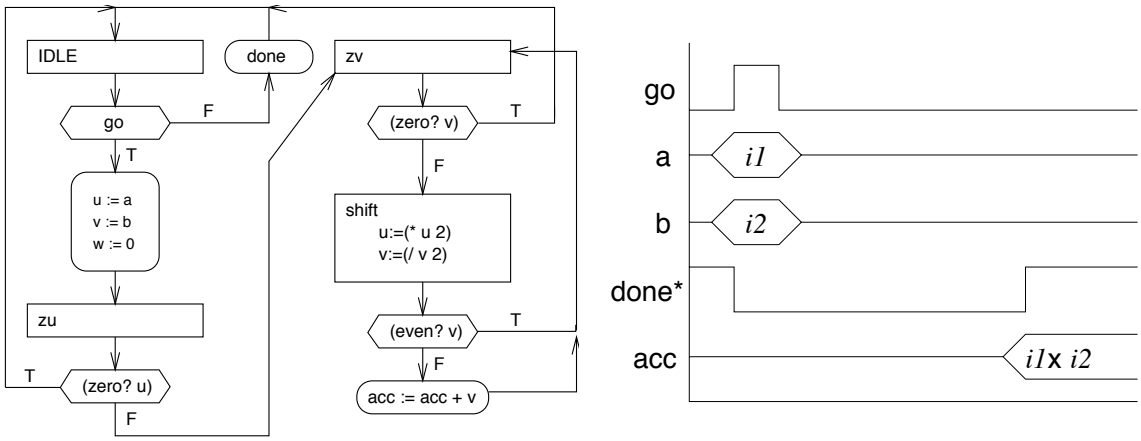
Chapter 6

System Factorization and Decomposition

System factorization is the organization of systems into architectural components. In the context of behavior tables this corresponds to their decomposition into a connection hierarchy in which behavior tables are the leaves. Factorization objectives include: isolating functions in need of further specification, encapsulating abstract data types, grouping complementary functionality, targeting known components, and separating system control from system data

A designer has three decisions to make when factoring a given system component: which functions and signals to factor, how to group functionality in the presence scheduling conflicts, and allocating communication signals between components. Behavior tables facilitate the process by collecting candidate terms into columns. Enhanced implementations use colorization (though only minimal support exists in Starfish) to further aid subject term selection. Some of these decisions are characterized as optimization problems; we can automate more of the process by solving these problems.

Three examples illustrate system factorization through application of the core table-algebra in Section 4.2.2 (p. 63). These examples decompose a behavior table



MULT: (go,a,b) → (done,acc)

go	state	state	u	v	acc	done
0	idle	idle	↯	↯	acc	1
1	idle	ztest	a	b	0	0
↯	zu	(if (zero? u) idle zv)	u	v	acc	0
↯	zv	(if (zero? v) idle shift)	u	v	acc	0
↯	shift	zv	(* u 2)	(/ v 2)	(if (even? v) acc (+ acc v))	0

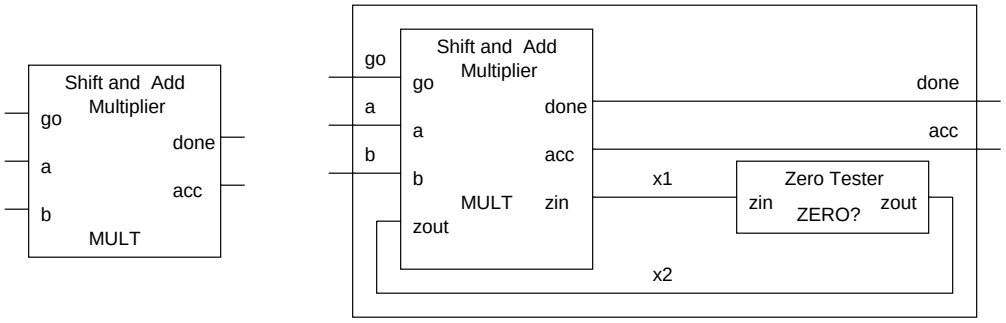
Figure 9: Algorithmic, protocol, and behavior table specification of a shift-and-add multiplier

describing a "shift-and-add" multiplier corresponding to the ASM chart [104] and timing diagram in Figure 9. Since the specification operates on mathematical integers, it "shifts" by division or multiplication by 2. The examples illustrate three broad classes of decompositions: single function factorization, multiple function factorization, and signal factorization. The core transformation rules affect system factorization at a fine level of granularity, often requiring hundreds of steps to produce a target result. The chapter ends with a description of higher level transformations and heuristics that automate many of the rote details to make factorization accessible with a small number of commands.

6.1 Single Function Factorization

The simplest system factorization separates instances of one particular function f from a sequential system S. The decomposition produces two tables: a trivial table F that unconditionally applies f to its inputs, and a modified system S' where instances of f are replaced with references to F's output signal and new combinational signals have been created to carry the arguments of f to F.

We illustrate this type of factorization by isolating the function `zero?` from the multiplier in Fig. 9. The block diagram view summarizes this architectural refinement. The connections hierarchy is maintained using the lambda expression below:



$$\begin{array}{lll} \lambda(go, a, b).(done, acc) \ \textit{where} & & \\ \quad (done, acc, x1) & = & MULT(go, a, b, x2) \\ \quad (x2) & = & ZERO?(x1) \end{array} \tag{69}$$

For the remainder of the examples, we assign I/O signals of separate blocks the same name to imply connection.

The *specification-set* defines the function(s) to be factored; it is $\{zero?\}$ for this illustration. Guiding selection of *subject terms*, the instances of functions from the specification-set the user intends to factor, the table below highlights all applications of `zero?`. We make no refinement to this initial selection of subject terms.

Inputs: (go, a, b)
Outputs: (done, acc)

go	state	state:Seq.	u:Seq.	v:Seq.	acc:Seq.	done:Comb.
false	idle	idle	#	#	acc	true
true	idle	zu	a	b	0	false
#	zu	(sel (zero? u) idle zv)	u	v	acc	false
#	zv	(sel (zero? v) idle shift)	u	v	acc	false
#	shift	zv	(* u 2)	(/ v 2)	(sel (even? v) acc (+ acc v))	false

(70)

Tabular notation conveys scheduling information because the actions of each row occur in the same step. Judicious choice of color scheme can identify scheduling collisions (i.e., rows with multiple subject terms) by using different colors to render the conflicting terms. This example has consistent scheduling because each use of `zero?` occurs in a different row; we display these terms with a green background. The factorization process has several subgoals:

Introduce "host signals" for inputs, outputs and internal signals of the zero tester component. In this case there is only one kind of input, a single integer, and one kind of output, a boolean. We introduce combinational signals to host the function application `zero?` and its inputs, `u` and `v`.

```
(add-act-col (make-sys-path) 'z_in "integer" 'comb)
(add-act-col (make-sys-path) 'z_out "boolean" 'comb)
```

Inputs: (go, a, b)
Outputs: (done, acc)

go	state	state:Seq.	u:Seq.	v:Seq.	acc:Seq.	done:Comb.	z_in:Comb.	z_out:Comb.
false	idle	idle	#	#	acc	true	#	#
true	idle	zu	a	b	0	false	#	#
#	zu	(sel (zero? u) idle zv)	u	v	acc	false	#	#
#	zv	(sel (zero? v) idle shift)	u	v	acc	false	#	#
#	shift	zv	(* u 2)	(/ v 2)	(sel (even? v) acc (+ acc v))	false	#	#

(71)

Populate these signals with term instantiation. The goal is to separate instances of `zero?` from the control specification. In this step, the instantiation rule commits copies of the subject terms to the host columns. The inputs, `u` and `v`, populate the input host, `z_in`, and the terms, `(zero? u)` and `(zero? v)`, populate the output host, `z_out`.

```
(specialize-term (make-sys-path) "z_in" '("#" "zu") "u")
(specialize-term (make-sys-path) "z_in" '("#" "zv") "v")
(specialize-term (make-sys-path) "z_out" '("#" "zu") "(zero? u)")
(specialize-term (make-sys-path) "z_out" '("#" "zv") "(zero? v)")
```

Inputs: (go, a, b)
Outputs: (done, acc)

go	state	state:Seq.	u:Seq.	v:Seq.	acc:Seq.	done:Comb.	z_in:Comb.	z_out:Comb.
false	idle	idle	#	#	acc	true	#	#
true	idle	zu	a	b	0	false	#	#
#	zu	(sel (zero? u) idle zv)	u	v	acc	false	u	(zero? u)
#	zv	(sel (zero? v) idle shift)	u	v	acc	false	v	(zero? v)
#	shift	zv	(* u 2)	(/ v 2)	(sel (even? v) acc (+ acc v))	false	#	#

(72)

Rewrite subject terms using the newly instantiated host signals. References to `z_out` replace the subject terms in the third and fourth rows using the combinational identification rule (folding). The inputs to `zero?` are replaced by references to `z_in`.

```
(apply-comb-ident '() '("#" "zu") "state" '(0) "z_out")
(apply-comb-ident '() '("#" "zv") "state" '(0) "z_out")
(apply-comb-ident '() '("#" "zu") "z_out" '(0) "z_in")
(apply-comb-ident '() '("#" "zv") "z_out" '(0) "z_in")
```

Inputs: (go, a, b)
Outputs: (done, acc)

go	state	state:Seq.	u:Seq.	v:Seq.	acc:Seq.	done:Comb.	z_in:Comb.	z_out:Comb.
false	idle	idle	#	#	acc	true	#	#
true	idle	zu	a	b	0	false	#	#
#	zu	(sel z_out idle zv)	u	v	acc	false	u	(zero? z_in)
#	zv	(sel z_out idle shift)	u	v	acc	false	v	(zero? z_in)
#	shift	zv	(* u 2)	(/ v 2)	(sel (even? v) acc (+ acc v))	false	#	#

(73)

Split system into two tables. This derivation splits the `z_out` signal from the remainder of the table. The resulting tables share the same decision table, and the minimal inputs necessary to bind all variable references in each behavior table. Automatically generated labels, `Shift and Add Multiplier 1` and `Shift and Add Multiplier 2`, identify the controller (large table) and zero-unit (small table) respectively.

```
(split-sys-table (make-sys-path) '((0 1 2 3 4 5) (6)))
```

Inputs: (a, z_out, b, go)
Outputs: (state, z_in, done, acc)

go	state	state:Seq.	u:Seq.	v:Seq.	acc:Seq.	done:Comb.	z_in:Comb.
false	idle	idle	#	#	acc	true	#
true	idle	zu	a	b	0	false	#
#	zu	(sel z_out idle zv)	u	v	acc	false	u
#	zv	(sel z_out idle shift)	u	v	acc	false	v
#	shift	zv	(* u 2)	(/ v 2)	(sel (even? v) acc (+ acc v))	false	#

(74)

Inputs: (go, a, b)
Outputs: (done, acc)

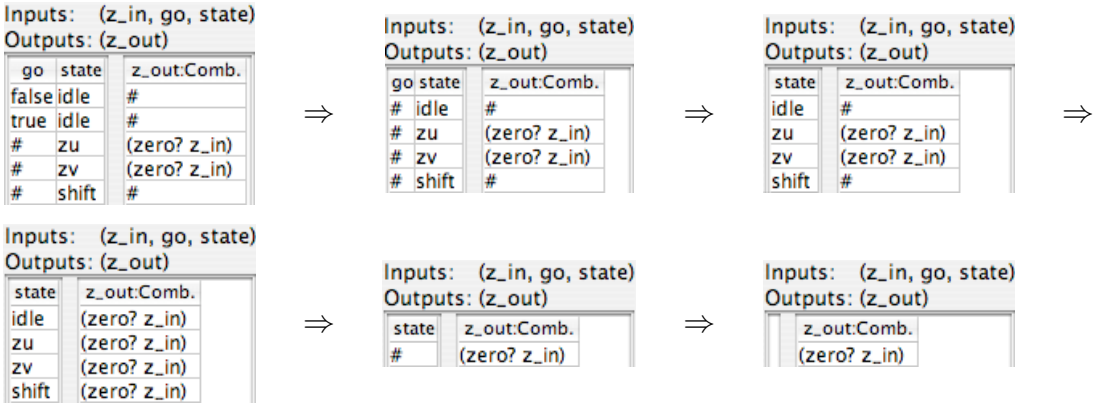
(state, z_in, done, acc) = Shift and Add Multiplier 1 (a, z_out, b, go)
(z_out) = Shift and Add Multiplier 2 (z_in, go, state)

Inputs: (z_in, go, state)
Outputs: (z_out)

go	state	z_out:Comb.
false	idle	#
true	idle	#
#	zu	(zero? z_in)
#	zv	(zero? z_in)
#	shift	#

Simplify decision table. The decision table in the `zero?` component obfuscates its role as a functional component. This derivation eliminates the decision table completely by instantiating the remaining #'s to match term instances, applying the decision instantiation rule to collapse the rows, and removing decision columns that contain only #'s. The transformations apply to the `zero?` behavior table, and the changes are summarized below:

```
(collapse-rows (make-sys-path 1) 0 '(0 1))
(remove-pred-col (make-sys-path 1) 0)
(specialize-term (make-sys-path 1) "z_out" '("idle") "(zero? z_in)")
(specialize-term (make-sys-path 1) "z_out" '("shift") "(zero? z_in)")
(collapse-rows (make-sys-path 1) 0 '(0 1 2 3))
(remove-pred-col (make-sys-path 1) 0)
```



(75)

Eliminate useless connecting signals. Now that the decision table has been eliminated, the input signals `go` and `state` are no longer necessary to close the expressions of the `zero?` table. The useless signal elimination rule dispenses with these two:

```
(remove-input-signal (make-input-addr '(1) 1))
(remove-input-signal (make-input-addr '(1) 1))
(remove-output-signal (make-output-addr '(0) 0))
```

Inputs: (a, z_out, b, go)
Outputs: (z_in, done, acc)

go	state	state:Seq.	u:Seq.	v:Seq.	acc:Seq.	done:Comb.	z_in:Comb.
false	idle	idle	#	#	acc	true	#
true	idle	zu	a	b	0	false	#
#	zu	(sel z_out idle zv)	u	v	acc	false	u
#	zv	(sel z_out idle shift)	u	v	acc	false	v
#	shift	zv	(* u 2)	(/ v 2)	(sel (even? v) acc (+ acc v))	false	#

(76)

Inputs: (go, a, b)
Outputs: (done, acc)

(z_in, done, acc) = Shift and Add Multiplier 1 (a, z_out, b, go)
(z_out) = Shift and Add Multiplier 2 (z_in)

Inputs: (z_in)
Outputs: (z_out)

z_out:Comb.
(zero? z_in)

This factorization encapsulates the function `zero?`, separating it from the initial specification. The resulting tables represent a "controller" and an "arithmetic unit". As factorizations accumulate, the architecture becomes more explicit and the table that began as a behavioral specification becomes a controller for the encapsulated components.

6.2 Factoring Multiple Functions

We can generalize factorization to permit the encapsulation of several functions. Tactically, this transformation follows the rule-application flow of the single function variant, but faces two important new issues: the desired function must be specified by some method, and the functions need not have the same input or output signatures. The first issue is easily solved by adding an instruction input to specify functionality. The second issue is one aspect of the much larger interface specification problem and requires guidance. In our factorization model, all the inputs are simultaneously

transferred to the resulting component, and the principal concern is optimization of input and output lines according to type; e.g., communication lines cannot be used alternately for integers and then booleans. In general, data transfer between components may occur serially through some multi-step protocol, as formulated by Rath's ISL [83]. However, protocol specification is a separate problem from architectural decomposition and Starfish does not address it further in the factorization process.

Arithmetic units and libraries commonly bundle zero testing and addition in the same components. This example's two-function factorization removes functions `zero?` and `+` from the initial specification. As specified, `zero?` returns a boolean and `+` returns an integer, so they cannot share a return channel. However, both functions accept integers, so they can share input channels; the designer specifies how to allocate sharing. Here, two input signals `in1` and `in2` host the inputs for `+`. The first signal, `in1`, carries inputs for `zero?`. Two separate output signals `out_+` and `out_z` communicate the results of the two subject functions. An instruction token, hosted by the signal `inst`, ranges over the enumerated type {`add, zero`}. Following the same pattern as before, the signal introduction and instantiation rules produce the input, output and subject signals for the factorization; combinational identification enables rewrites of subject terms using host signals. The tables below color the `zero?` subject terms and corresponding host signals in green, and the `+` subject term with corresponding host signals in cyan.

```
(add-act-col (make-sys-path) 'in1 "integer" 'comb)
(add-act-col (make-sys-path) 'in2 "integer" 'comb)
(add-act-col (make-sys-path) 'inst "f-sel" 'comb)
(add-act-col (make-sys-path) 'out_+ "integer" 'comb)
(add-act-col (make-sys-path) 'out_z "boolean" 'comb)
(specialize-term (make-sys-path) "in1" '("#" "zu") "u")
(specialize-term (make-sys-path) "in1" '("#" "zv") "v")
(specialize-term (make-sys-path) "in1" '("#" "shift") "acc")
(specialize-term (make-sys-path) "in2" '("#" "shift") "v")
(specialize-term (make-sys-path) "inst" '("#" "zu") "zero")
(specialize-term (make-sys-path) "inst" '("#" "zv") "zero")
(specialize-term (make-sys-path) "inst" '("#" "shift") "add")
(specialize-term (make-sys-path) "out_z" '("#" "zu") "(zero? u)")
(specialize-term (make-sys-path) "out_z" '("#" "zv") "(zero? v)")
(specialize-term (make-sys-path) "out_+" '("#" "shift") "(+ acc v)")
```

Inputs: (go, a, b)
Outputs: (done, acc)

go	state	state:Seq.	u:Seq.	v:Seq.	acc:Seq.	done:Comb.	in1:Comb.	in2:Comb.	inst:Comb.	out_+:Comb.	out_z:Comb.
false	idle	idle	#	#	acc	true	#	#	#	#	#
true	idle	zu	a	b	0	false	#	#	#	#	#
#	zu	(sel (zero? u) idle zv)	u	v	acc	false	u	#	zero	#	(zero? u)
#	zv	(sel (zero? v) idle shift)	u	v	acc	false	v	#	zero	#	(zero? v)
#	shift	zv	(* u 2)	(/ v 2)	(sel (even? v) acc (+ acc v))	false	acc	v	add	(+ acc v)	#

(77)

Combinational identification integrates the new signals `out_z`, `out_+`, `in1`, and `in2` with the actions for `state`, `acc`, `out_z`, and `out_+`:

```
(apply-comb-ident '() '("#" "zu") "state" '(0) "out_z")
(apply-comb-ident '() '("#" "zv") "state" '(0) "out_z")
(apply-comb-ident '() '("#" "shift") "acc" '(2) "out_+")
(apply-comb-ident '() '("#" "zu") "out_z" '(0) "in1")
(apply-comb-ident '() '("#" "zv") "out_z" '(0) "in1")
(apply-comb-ident '() '("#" "shift") "out_+" '(0) "in1")
(apply-comb-ident '() '("#" "shift") "out_+" '(1) "in2")
```

Inputs: (go, a, b)
Outputs: (done, acc)

go	state	state:Seq.	u:Seq.	v:Seq.	acc:Seq.	done:Comb.	in1:Comb.	in2:Comb.	inst:Comb.	out_+:Comb.	out_z:Comb.
false	idle	idle	#	#	acc	true	#	#	#	#	#
true	idle	zu	a	b	0	false	#	#	#	#	#
#	zu	(sel out_z idle zv)	u	v	acc	false	u	#	zero	#	(zero? in1)
#	zv	(sel out_z idle shift)	u	v	acc	false	v	#	zero	#	(zero? in1)
#	shift	zv	(* u 2)	(/ v 2)	(sel (even? v) acc out_+)	false	acc	v	add	(+ in1 in2)	#

(78)

At this point, the derivation script anticipates the decision table of the two operation table by encoding the decisions into selector terms. The target table uses the `inst` signal and the tokens `add` and `zero` to either return `(+ in1 in2)` on the output signal `out_+` or `(zero? in1)` on the output signal `out_z`, respectively. The algebraic identities of selection over {`add, zero`} let us replace `x` with `(sel add x #)` or `(sel zero # x)`. Applying these identities to the terms of the subject signals yields the table below. Once transformed, we instantiate the unspecified signals of the output columns to create uniform entries.

```
(expand-to-sel (make-act-subterm-addr '() 9 2 '()) "zero")
(apply-comb-ident '() '("#" "zu") "out_z" '(0) "inst")
(expand-to-sel (make-act-subterm-addr '() 9 3 '()) "zero")
(apply-comb-ident '() '("#" "zv") "out_z" '(0) "inst")
(specialize-term (make-sys-path)
                 "out_z" '("false" "idle") "(sel inst # (zero? in1))")
(specialize-term (make-sys-path)
                 "out_z" '("true" "idle") "(sel inst # (zero? in1))")
(specialize-term (make-sys-path)
                 "out_z" '("#" "shift") "(sel inst # (zero? in1))")
(expand-to-sel (make-act-subterm-addr '() 8 4 '()) "add")
(apply-comb-ident '() '("#" "shift") "out_+" '(0) "inst")
(specialize-term (make-sys-path)
                 "out_+" '("false" "idle") "(sel inst (+ in1 in2) #)")
(specialize-term (make-sys-path)
                 "out_+" '("true" "idle") "(sel inst (+ in1 in2) #)")
(specialize-term (make-sys-path)
                 "out_+" '("#" "zu") "(sel inst (+ in1 in2) #)")
(specialize-term (make-sys-path)
                 "out_+" '("#" "zv") "(sel inst (+ in1 in2) #)")
```

Inputs: (go, a, b)
Outputs: (done, acc)

go	state	state:Seq.	u:Seq.	v:Seq.	acc:Seq.	done:Comb.	in1:Comb.	in2:Comb.	inst:Comb.	out_+:Comb.	out_z:Comb.
false	idle	idle	#	#	acc	true	#	#	#	(sel inst (+ in1 in2) #)	(sel inst # (zero? in1))
true	idle	zu	a	b	0	false	#	#	#	(sel inst (+ in1 in2) #)	(sel inst # (zero? in1))
#	zu	(sel out_z idle zv)	u	v	acc	false	u	#	zero	(sel inst (+ in1 in2) #)	(sel inst # (zero? in1))
#	zv	(sel out_z idle shift)	u	v	acc	false	v	#	zero	(sel inst (+ in1 in2) #)	(sel inst # (zero? in1))
#	shift	zv	(* u 2)	(/ v 2)	(sel (even? v) acc out_+)	false	acc	v	add	(sel inst (+ in1 in2) #)	(sel inst # (zero? in1))

(79)

The next step splits the signals `out_+` and `out_z` from the main table. Since the actions are identical across all rows for both of these signals, the decision table inherited from its parent description completely collapses through repeated application

of the decision instantiation and decision elimination rules. With the decision table headings gone, it is safe to remove the inputs `go` and `state`. These transformations follow the last three steps of single function factorization.

```
(split-sys-table (make-sys-path) '((0 1 2 3 4 5 6 7) (8 9)))
(collapse-rows (make-sys-path 1) 0 '(0 1))
(remove-pred-col (make-sys-path 1) 0)
(collapse-rows (make-sys-path 1) 0 '(0 1 2 3))
(remove-pred-col (make-sys-path 1) 0)
(remove-input-signal (make-input-addr '(1) 3))
(remove-input-signal (make-input-addr '(1) 3))
(remove-output-signal (make-output-addr '(0) 0))
```

Inputs: (out_+, a, out_z, b, go)
Outputs: (in2, inst, in1, done, acc)

go	state	state:Seq.	u:Seq.	v:Seq.	acc:Seq.	done:Comb.	in1:Comb.	in2:Comb.	inst:Comb.
false	idle	idle	#	#	acc	true	#	#	#
true	idle	zu	a	b	0	false	#	#	#
#	zu	(sel out_z idle zv)	u	v	acc	false	u	#	zero
#	zv	(sel out_z idle shift)	u	v	acc	false	v	#	zero
#	shift	zv	(* u 2)	(/ v 2)	(sel (even? v) acc out_+)	false	acc	v	add

Inputs: (go, a, b)
Outputs: (done, acc)

(in2, inst, in1, done, acc) = Shift and Add Multiplier 1 (out_+, a, out_z, b, go)
(out_z, out_+) = Shift and Add Multiplier 2 (in1, inst, in2)

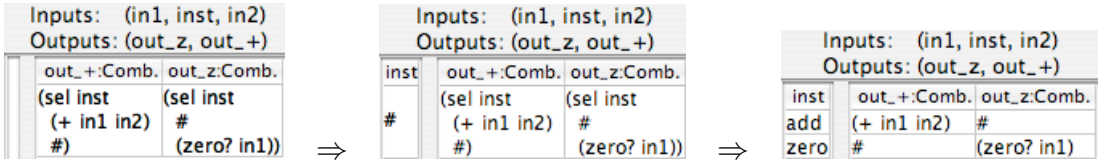
Inputs: (in1, inst, in2)
Outputs: (out_z, out_+)

out_+:Comb.	out_z:Comb.
(sel inst (+ in1 in2) #)	(sel inst # (zero? in1))

(80)

The factorization finishes by introducing `inst` as a decision table heading, and instantiating along this term. This transformation transfers the work of the selectors to explicit enumeration in the decision table.

```
(add-pred-col (make-sys-path 1) "inst")
(expand-row (make-sys-path 1) '("#") "inst")
```



(81)

6.3 Signal Factorization

Factorization in the previous examples creates combinational signals to host subject functions and inputs, and then splits them off into a table. Splitting off state holding sequential signals is another way to elaborate architecture. Sometimes this is desirable because the subject states operate on abstract data types, and the designer needs to isolate the abstraction from the remainder of the system. Other times, a particular signal is a natural target for an existing component implementation.

Beginning this example where the one from Sec. 6.2 ends, it encapsulates signals `u` and `v`. Intuitively, factoring these signals yields a pair of shift registers, a common component. The remaining component is the essence of a controller, containing no function applications (except for `even?` which can be simplified to a boolean test). This decomposition factors `u` and `v`, expands selectors into the decision table, and then flattens the table hierarchy to place the three tables in the same level.

As before, the decomposition sends instructions through a new signal, `inst1,` to manage operations on the state. The specification file declares an enumerated type, `{init, hold, shft}` to reference the kinds of state changes that can occur. The table below colors the three different operations on the state.

```
(add-act-col (make-sys-path 0) 'inst1 "mshift-tok" 'comb)
(specialize-term (make-sys-path 0) "inst1" '("true" "idle") "init")
(specialize-term (make-sys-path 0) "inst1" '("#" "zu") "hold")
(specialize-term (make-sys-path 0) "inst1" '("#" "zv") "hold")
(specialize-term (make-sys-path 0) "inst1" '("#" "shift") "shft")
```

Inputs: (out_+, a, out_z, b, go)
Outputs: (in2, inst, in1, done, acc)

go	state	state:Seq.	u:Seq.	v:Seq.	acc:Seq.	done:Comb.	in1:Comb.	in2:Comb.	inst:Comb.	inst1:Comb.
false	idle	idle	#	#	acc	true	#	#	#	#
true	idle	zu	a	b	0	false	#	#	#	init
#	zu	(sel out_z idle zv)	u	v	acc	false	u	#	zero	hold
#	zv	(sel out_z idle shift)	u	v	acc	false	v	#	zero	hold
#	shift	zv	(* u 2)	(/ v 2)	(sel (even? v) acc out_+)	false	acc	v	add	shft

(82)

Like the previous examples, we anticipate the decision table of the new table with uniform selection terms over the subject signals, `u` and `v`. This is done through repeated application of selector identities keyed by the instruction type, instantiating unspecified terms, and combinational identification.

```
;; Prepping u column
(expand-to-sel (make-act-subterm-addr '(0) 1 1 '()) "init")
(expand-to-sel (make-act-subterm-addr '(0) 1 2 '()) "hold")
(expand-to-sel (make-act-subterm-addr '(0) 1 3 '()) "hold")
(expand-to-sel (make-act-subterm-addr '(0) 1 4 '()) "shft")
(specialize-term (make-sys-path 0) "u" '("true" "idle") "u" '(2))
(specialize-term (make-sys-path 0) "u" '("true" "idle") "(* u 2)" '(3))
(specialize-term (make-sys-path 0) "u" '("#" "zu") "a" '(1))
(specialize-term (make-sys-path 0) "u" '("#" "zu") "(* u 2)" '(3))
(specialize-term (make-sys-path 0) "u" '("#" "zv") "a" '(1))
(specialize-term (make-sys-path 0) "u" '("#" "zv") "(* u 2)" '(3))
(specialize-term (make-sys-path 0) "u" '("#" "shift") "a" '(1))
(specialize-term (make-sys-path 0) "u" '("#" "shift") "u" '(2))
(apply-comb-ident '(0) '("true" "idle") "u" '(0) "inst1")
(apply-comb-ident '(0) '("#" "zu") "u" '(0) "inst1")
(apply-comb-ident '(0) '("#" "zv") "u" '(0) "inst1")
(apply-comb-ident '(0) '("#" "shift") "u" '(0) "inst1")
(specialize-term (make-sys-path 0) "u" '("false" "idle")
  "(sel inst1 a u (* u 2))")

;; Prepping v column
(expand-to-sel (make-act-subterm-addr '(0) 2 1 '()) "init")
(expand-to-sel (make-act-subterm-addr '(0) 2 2 '()) "hold")
(expand-to-sel (make-act-subterm-addr '(0) 2 3 '()) "hold")
(expand-to-sel (make-act-subterm-addr '(0) 2 4 '()) "shft")
(specialize-term (make-sys-path 0) "v" '("true" "idle") "v" '(2))
(specialize-term (make-sys-path 0) "v" '("true" "idle") "(/ v 2)" '(3))
(specialize-term (make-sys-path 0) "v" '("#" "zu") "b" '(1))
(specialize-term (make-sys-path 0) "v" '("#" "zu") "(/ v 2)" '(3))
(specialize-term (make-sys-path 0) "v" '("#" "zv") "b" '(1))
(specialize-term (make-sys-path 0) "v" '("#" "zv") "(/ v 2)" '(3))
(specialize-term (make-sys-path 0) "v" '("#" "shift") "b" '(1))
(specialize-term (make-sys-path 0) "v" '("#" "shift") "v" '(2))
(apply-comb-ident '(0) '("true" "idle") "v" '(0) "inst1")
(apply-comb-ident '(0) '("#" "zu") "v" '(0) "inst1")
(apply-comb-ident '(0) '("#" "zv") "v" '(0) "inst1")
(apply-comb-ident '(0) '("#" "shift") "v" '(0) "inst1")
(specialize-term (make-sys-path 0) "v" '("false" "idle")
  "(sel inst1 b v (/ v 2))")
```

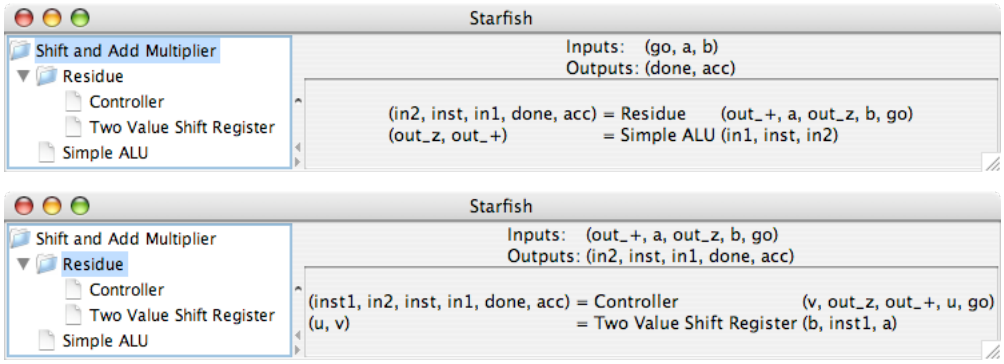
go	state	state:Seq.	u:Seq.	v:Seq.	acc:Seq.	done:Comb.	in1:Comb.	in2:Comb.	inst:Comb.	inst1:Comb.
false	idle	idle	(sel inst1 a u (* u 2))	(sel inst1 b v (/ v 2))	acc	true	#	#	#	#
true	idle	zu	(sel inst1 a u (* u 2))	(sel inst1 b v (/ v 2))	0	false	#	#	#	init
#	zu	(sel out_z idle zv)	(sel inst1 a u (* u 2))	(sel inst1 b v (/ v 2))	acc	false	u	#	zero	hold
#	zv	(sel out_z idle shift)	(sel inst1 a u (* u 2))	(sel inst1 b v (/ v 2))	acc	false	v	#	zero	hold
#	shift	zv	(sel inst1 a u (* u 2))	(sel inst1 b v (/ v 2))	(sel (even? v) acc out_+)	false	acc	v	add	shft

(83)

The subsequent steps mirror the previous examples as well. The decomposition rule splits the subject signals from the controller table. The decision generalization (opposite of instantiation) elimination rules remove the legacy decision table, making the input signals `go` and `state` defunct; they are subsequently removed. Decision introduction and instantiation move the instruction-keyed selectors into the decision table. The resulting hierarchy is two levels deep, one for each decomposition.

```
;; Splitting Table
(split-sys-table (make-sys-path 0) '((0 3 4 5 6 7 8) (1 2)))

;; Collapsing old DT
(collapse-rows (make-sys-path 0 1) 0 '(0 1))
(collapse-rows (make-sys-path 0 1) 1 '(0 1 2 3))
(remove-pred-col (make-sys-path 0 1) 0)
(remove-pred-col (make-sys-path 0 1) 0)
```

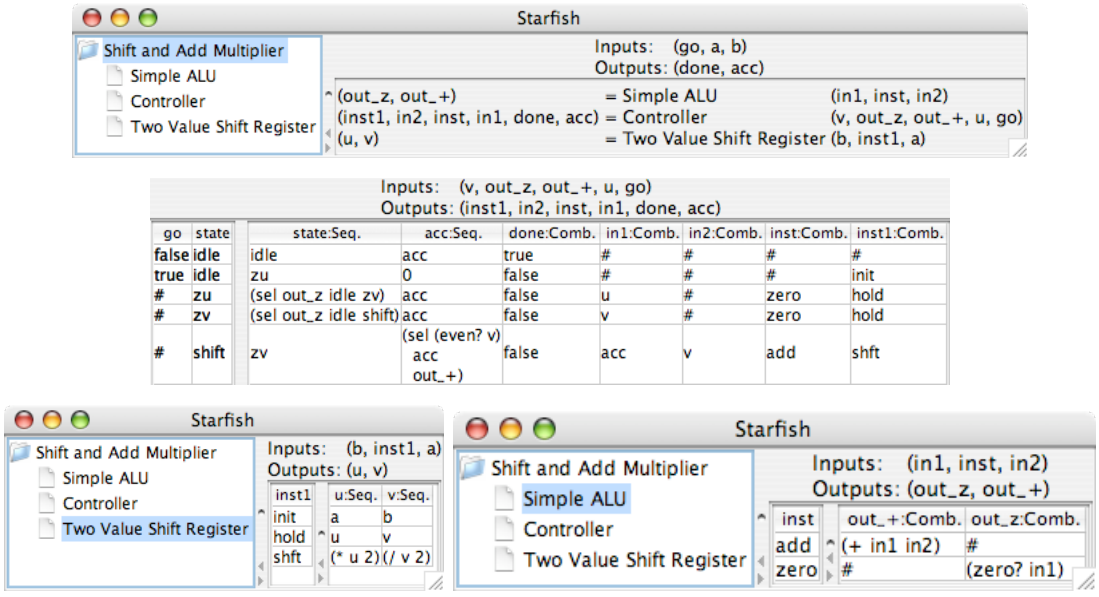
```
;; Eliminating defunct inputs and outputs
(remove-input-signal (make-input-addr '(0 1) 4))
(remove-input-signal (make-input-addr '(0 1) 3))
(remove-output-signal (make-output-addr '(0 0) 0))

;; Expanding DT over 'inst1'
(add-pred-col (make-sys-path 0 1) "inst1")
(expand-row (make-sys-path 0 1) '("#") "inst1")
```



(84)

The next transformation flattens this hierarchy to connect all three behavior tables at the same level. The tables below show the complete system view after hierarchy manipulation.

```
(flatten-sys-node (make-sys-path 0))
```



(85)

The last step expands selector terms in the action table into the decision table. The resulting form explicitly represents a Mealy machine. The decision table encodes the next state function without any black-box functional logic; `even?` is just a projection of an integer's least significant bit in target implementations. Action updates are all constant terms, with the sequential signal `state` holding the control state, the combinational signals defining the output tokens, and the `acc` signal behaving as part of the data path.

```
(add-pred-col (make-sys-path 0 0) "out_z")
(expand-row (make-sys-path 0 0) '("#" "zu" "#") "out_z")
(expand-row (make-sys-path 0 0) '("#" "zv" "#") "out_z")
(add-pred-col (make-sys-path 0 0) "(even? v)")
(expand-row (make-sys-path 0 0) '("#" "shift" "#" "#") "(even? v)")
```

Inputs: (v, out_z, out_+, u, go)
Outputs: (inst1, in2, inst, in1, done, acc)

go	state	out_z	(even? v)	state:Seq.	acc:Seq.	done:Comb.	in1:Comb.	in2:Comb.	inst:Comb.	inst1:Comb.
false	idle	#	#	idle	acc	true	#	#	#	#
true	idle	#	#	zu	0	false	#	#	#	init
#	zu	true	#	idle	acc	false	u	#	zero	hold
#	zu	false	#	zv	acc	false	u	#	zero	hold
#	zv	true	#	idle	acc	false	v	#	zero	hold
#	zv	false	#	shift	acc	false	v	#	zero	hold
#	shift	#	true	zv	acc	false	acc	v	add	shft
#	shift	#	false	zv	out_+	false	acc	v	add	shft

(86)

6.4 Increasing Automation

Many of the rule transitions from the preceding factorization examples are entirely predictable and automatable. For example, converting a column of actions to a uniform selector keyed to an instruction signal consumes repeated selector identities, subterm instantiations, and combinational identifications. Other decisions, such as allocation of input and output signals, are difficult to optimize in general, however the interactive process benefits from suboptimal solutions. Starfish facilitates system factorizations with three high level transformations that automate many of the tedious details, heuristically solve the signal allocation problem, and encourage interactive refinement of suboptimal solutions: `expand-apps`, `permute-comb-signals`, and `factor-signals`.

The `expand-apps` transformation prepares a table for factorization by introducing an initializing input and output host signals based on the specification-set. It employs heuristics that appear to work reasonably well in most cases we have encountered. Of course, the designer can manually intervene if the algorithm produces poor results.

The purpose of this transformation is to create a plausible first result for further refinement by designer interaction.

The set is specified by a list of function addresses and a list of signals—in case of a signal factorization. In the case of pure function factorization the transformation examines each function subterm according to the following procedure:

1. For each input parameter i to f
   (a) Is there an input host signal of type $\tau(i)$?
      i. If yes, let arg_x be this open signal.
      ii. If no, introduce a new combinational signal of type $\tau(i)$, arg_x.
   (b) Instantiate the input host, arg_x to the value of i.
   (c) Combinationally replace $f(\ldots, i, \ldots)$ with $f(\ldots, arg_x, \ldots)$.
2. Is there an open output host signal of type $\tau_{out}(f)$?
   (a) If yes, let app_x be this open signal.
   (b) If no, introduce a new combinational signal of type $\tau_{out}(f)$, app_x.
3. Instantiate the output host, app_x to the value of $f(arg_{i_1}, \ldots, arg_{i_n})$.
4. Combinationally replace the original subject term $f(arg_{i_1}, \ldots, arg_{i_n})$ with app_x.

This is a very simple algorithm, and the quality of its results depend on the order of subject term consideration and which of the eligible host signals is selected. Input parameters are specified with subterm addresses. The specification list could become stale, if the intermediate transformations executed by `expand-apps` alter the parent term of a specification address prior to allocating an output register.

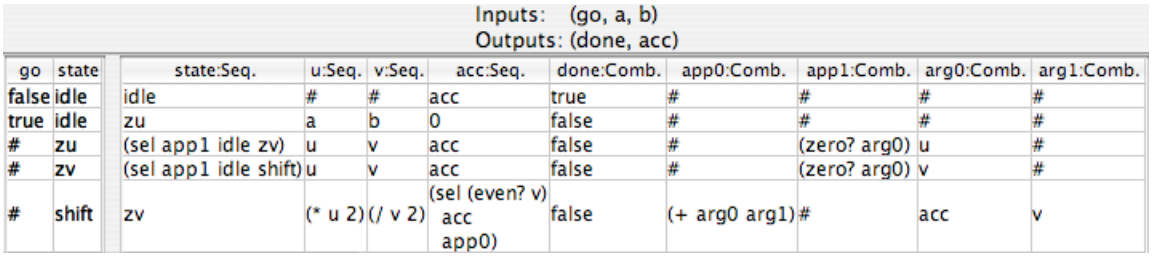
Consider one pathological case, where one subject term, $g(x)$, is a subterm of another subject term, $f(g(x))$. Suppose the address for f is given by the path `'(1 0 2)` and the path for g is given by `'(1 0 2 0)`. Suppose the algorithm processes

the reference to f first, placing $g(x)$ into a host input signal, arg_1, $f(arg_1)$ into host output signal, app_1, and then replaces $f(g(x))$ with app_1. Now the reference `'(1 0 2 0)` is stale because `'(1 0 2)` is an address for the term app_1 which no longer has $g(x)$ as a subterm. Either the subject term address must be updated to reference the post allocation table, *or* the algorithm should have expanded the child term prior to the parent term. Since expansion of child terms does not affect the validity of parent references, the algorithm first sorts the function address by term depth.

When more than one correctly typed output host signal is available, the algorithm chooses the "oldest" one; signal names, arg_i, reflect the time of their introduction, so the algorithm chooses the one with the smallest i. Under this policy, there is some benefit to considering subject terms in groups of functions. So within the child-to-parent ordering, the subject terms are alphabetically ordered. Input host signals are allocated using the oldest available signal.

The table below shows the effect of `expand-apps` on the shift and add multiplier. It prepares the specification for factorization by expanding `+` and `zero?` into input and output host signals. Rather than specify subject terms with their absolute addresses, this command invokes `search-term-action-table` to return the addresses of all applications of `+` or `zero?`.

```
(expand-apps
  '()
  (search-term-action-table
    (fish-state->sys (hist->state sess-log))
    (make-sys-path)
    (lambda (t)
      (match-dt t
        ([(app (func :: func name)) ((name))]
         (memq name '(+ zero?)))
        (else #f)))))
```



Inputs: (go, a, b)
Outputs: (done, acc)

go	state	state:Seq.	u:Seq.	v:Seq.	acc:Seq.	done:Comb.	app0:Comb.	app1:Comb.	arg0:Comb.	arg1:Comb.
false	idle	idle	#	#	acc	true	#	#	#	#
true	idle	zu	a	b	0	false	#	#	#	#
#	zu	(sel app1 idle zv)	u	v	acc	false	#	(zero? arg0)	u	#
#	zv	(sel app1 idle shift)	u	v	acc	false	#	(zero? arg0)	v	#
#	shift	zv	(* u 2)	(/ v 2)	(sel (even? v) acc app0)	false	(+ arg0 arg1)	#	acc	v

(87)

The `permute-comb-signals` transformation allows the designer to adjust allocation of host signals. Suppose that transposing the usage of `arg0` and `arg1` in `app0`'s `+` operation saves resources at lower levels because it reduces `arg0`'s input space from {`u`,`v`,`acc`} to {`u`,`v`}. The `permute-comb-signals` transformation takes a list of addresses that reference similarly typed host input (or output) terms in the same row, and performs a cyclic permutation on their usage following the order of the address list; e.g., lists of length two transpose their subjects and also swap references to these signals within the row, while lists of length greater than two move the term from one signal in the list to the next, looping at the end of the list. The following table shows its effect on the arguments to `+` in the bottom row (colorization added for emphasis):

```
(permute-comb-signals
 (list (make-act-subterm-addr (make-sys-path) 7 4 '())
       (make-act-subterm-addr (make-sys-path) 8 4 '())))
```

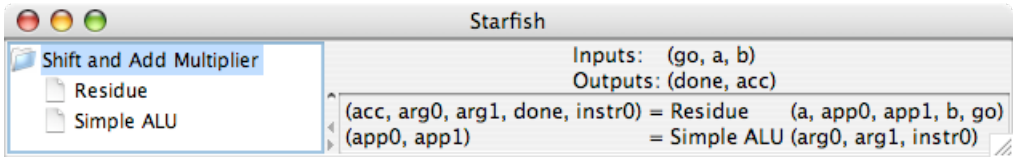
Inputs: (go, a, b)
Outputs: (done, acc)

go	state	state:Seq.	u:Seq.	v:Seq.	acc:Seq.	done:Comb.	app0:Comb.	app1:Comb.	arg0:Comb.	arg1:Comb.
false	idle	idle	#	#	acc	true	#	#	#	#
true	idle	zu	a	b	0	false	#	#	#	#
#	zu	(sel app1 idle zv)	u	v	acc	false	#	(zero? arg0)	u	#
#	zv	(sel app1 idle shift)	u	v	acc	false	#	(zero? arg0)	v	#
#	shift	zv	(* u 2)	(/ v 2)	(sel (even? v) acc app0)	false	(+ arg1 arg0)	#	v	acc

(88)

After adjusting host signal allocation with `permute-comb-signals`, the table is ready for decomposition. The `factor-signals` transformation takes a system address pointing to a behavior table, and a list of signal names to encapsulate; names for the resulting child nodes are also specified here. First it determines the number of unique actions (ignoring rows composed of entirely unspecified terms) relative to the list of signals. For example, when factoring the signals `app0` and `app1` from table (88), there are two unique actions: {`#, (zero? arg0)`} and {`(+ arg1 arg0), #`}. Then the transformation introduces a combinational instruction signal over a sufficiently large bit vector to "key" the different actions. In this case, only one bit is necessary to capture the two actions. In general, the number of actions is not a power of two, so there will be some unassigned branches in the resulting selectors. The transformation automates the remaining selector homogenization, decomposition, and decision table simplification steps from the previous factorization examples.

```
(factor-signals (make-sys-path) '(app0 app1) "Residue" "Simple ALU")
```



Inputs: (a, app0, app1, b, go)
Outputs: (acc, arg0, arg1, done, instr0)

go	state	state:Seq.	u:Seq.	v:Seq.	acc:Seq.	done:Comb.	arg0:Comb.	arg1:Comb.	instr0:Comb.
false	idle	idle	#	#	acc	true	#	#	#
true	idle	zu	a	b	0	false	#	#	#
#	zu	(sel app1 idle zv)	u	v	acc	false	u	#	0b0
#	zv	(sel app1 idle shift)	u	v	acc	false	v	#	0b0
#	shift	zv	(* u 2)	(/ v 2)	(sel (even? v) acc app0)	false	v	acc	0b1

(89)

Inputs: (arg0, arg1, instr0)
Outputs: (app0, app1)

instr0	app0:Comb.	app1:Comb.
0b0	#	(zero? arg0)
0b1	(+ arg1 arg0)	#

Chapter 7

Serialization

Behavior tables often apply many functions in their update action terms. There are several practical reasons why evaluating these actions as simultaneous events is infeasible: the target architecture may not have enough functional units to support the simultaneous computation, the data type implementation may only support one signal access per clock cycle, or a cooperating process requires multiple steps to consume and produce the necessary data. Serialization is one technique for specifying more finely-grained updates to a behavior table.

Starfish's serialization tables provide an interactive interface for producing a *schedule*—the result of high-level synthesis (HLS) applied to data-flow graphs (Figure 10). Given a graph, HLS algorithms attempt to fit the number of registers, functional units and control steps within specified requirements by partitioning the graph nodes into multisets—multiple nodes of the same operation may occupy a multiset. This partition is linearly ordered such that all required inputs for a multiset are produced by a preceding multiset. A controller simultaneously executes the nodes of each partition in order. Values that cross partition boundaries must be held in registers; this determines *register allocation*, the number of required registers for the schedule. The

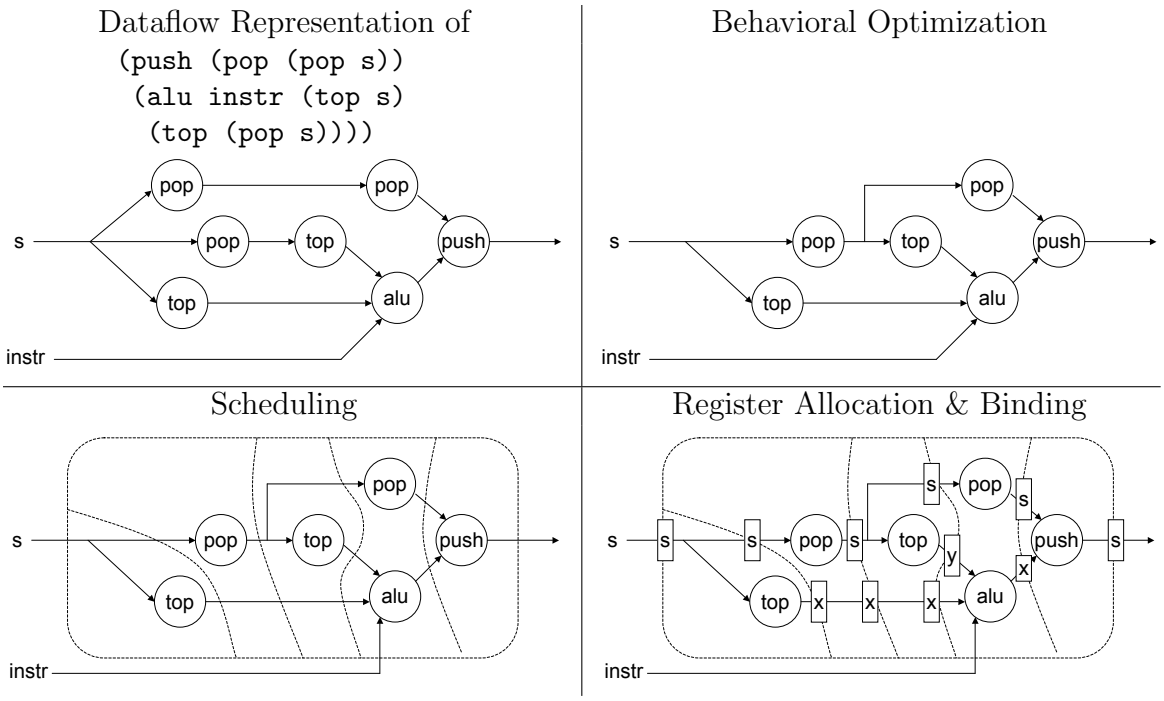


Figure 10: High-level synthesis objectives.

multiset union of partitions determines *function-unit allocation*, the required number of each kind of unit. *Binding*, the assignment of specific signals to registers or functional units, is not completely determined by the schedule but is responsible for communication costs (e.g., wires and fanout limits) in between registers and units.

Starfish facilitates the interactive specification of behavioral optimizations, scheduling, allocation and binding with *serialization tables* (Figure 11). Starfish terms express a data-flow graph where its functions are functional units and its variables are signals. The rows of a serialization table show signal updates (left) and their partial linear evaluation (right); the columns represent sequential (but not combinational) signals. The table's last row specifies the evaluation requirements for each signal. Serialization tables represent register allocation with their columns and register binding

s	x	y
s	(top s)	#
(pop s)	x	#
s	x	(top s)
(pop s)	(alu instr x y)	#
(push s x)	#	#
#	#	#

s	x	y
s	(top s)	#
(pop s)	(top s)	#
(pop s)	(top s)	(top (pop s))
(pop (pop s))	(alu instr (top s) (top (pop s)))	#
(push (pop (pop s)) (alu instr (top s) (top (pop s))))	#	#
(push (pop (pop s)) (alu instr (top s) (top (pop s))))	#	#

Figure 11: Serialization table representing the same result as Figure 10.

with term placement inside a column. The rows of the serialization table form the schedule or partition. Functional allocation is determined by the multiset union of operators over the rows; subsequent system factorization makes the allocation explicit by separating functions from term expressions.

7.1 How it works

Serialization specifies the evaluation order and storage of intermediate results for a behavior table's sequential signals. The result is a linear sequence of guards in the decision table that govern the control flow and a corresponding sequence of actions in the action table whose cumulative evaluation equals the pre-serialized signal actions.

Example 7.1: Figures 10 and 11 shows a valid serialization for the term:

$$\begin{array}{l}\text{(push (pop (pop s))}\\ \text{(alu instr (top s) (top (pop s))))}\end{array} \tag{90}$$

The stack calculator behavior table uses (90) to update `s` in case `alu-op`. The schedule developed in Figure 11 performs one stack access per step in anticipation of a single ported stack memory. The ALU operation is scheduled in parallel with the last `pop`. The schedule introduces two registers (or sequential signals) `x` and `y` to hold the intermediate integer values. Incorporating the schedule into the specification table produces the following change:

Inputs: (instr, a)
Outputs: (res)

(inst-cat instr)	res:Comb.	s:Seq.
psh-op	(top s)	(push s a)
drp-op	(top s)	(pop s)
alu-op	(top s)	(push (pop (pop s)) (alu instr (top s) (top (pop s))))

⇒

Inputs: (instr, a)
Outputs: (res)

ser	(inst-cat instr)	res:Comb.	s:Seq.	x:Seq.	y:Seq.
#	psh-op	(top s)	(push s a)	#	#
#	drp-op	(top s)	(pop s)	#	#
0	alu-op	(top s)	s	(top s)	#
1	alu-op	(top s)	(pop s)	x	#
2	alu-op	(top s)	s	x	(top s)
3	alu-op	(top s)	(pop s)	(alu instr x y)	#
4	alu-op	(top s)	(push s x)	#	#

The intuitive signal traces are shown below for this table:

$$\begin{array}{r|rrr|rrrrrrrrr|l}
inst= & (psh, & psh, & add, & \multicolumn{9}{c|}{\text{Hold value of } inst,} & psh, & \ldots) \\
a = & (\ 5, & 7, & \#, & \multicolumn{9}{c|}{\text{Hold value of } a,} & 25, & \ldots) \\
\hline
res = & (\ 0, & 5, & 7, & 7, & 5, & 5, & 0, & & & & 12, & \ldots) \\
s = & (\{0\}, & \{0,5\}, & \{0,5,7\}, & \{0,5,7\}, & \{0,5\}, & \{0,5\}, & \{0\}, & & & & \{0,12\}, & \ldots) \\
x = & (\ \#, & \#, & \#, & 7, & 7, & 7, & 12, & & & & \#, & \ldots) \\
y = & (\ \#, & \#, & \#, & \#, & \#, & 5, & \#, & & & & \#, & \ldots)
\end{array} \tag{91}$$

The first two streams are inputs. Vertical lines show externally observable synchronization events. The first two steps proceed as usual. The third synchronization step begins a sequence of "internal" transitions; only the last step in this sequence is externally observable. The input values of the previous step event hold for first

4 internal transitions. The input stream resumes in the fifth and final step of the schedule.

In illustrating the intuitive serialization semantics, Example 7.1 presents one possible accounting approach for stretched evaluation over multiple steps: assert that only the last step is externally visible. This corresponds to the Milner's *weak bisimulation* [71] notion which conceals a process's internal transitions.

The other approach is to make the new transitions externally visible and require the external agents to change their timing expectations. This could be a problem for agents expecting a result in a prescribed number of transitions. However, many specifications abstract these timing variances with interface protocols; e.g., the shift-and-add multiplier in Figure 9 indicates when it's working and when the result is ready with the `done` signal. Adding extra cycles to the to the non-`idle` states delays the `done` flag; external agents adhering to the protocol will have no need to change.

In either approach, serialization and factorization complicate the accounting. If one assumes that the transitions are invisible to external observers, what happens when one wants to factor a serialized behavior table? Clearly the factored components, such as ALUs and random-access storage, will be aware of the internal transitions. The natural solution is to define a *timing scope* (Figure 12) which specifies the boundary between visibility of internal and external transitions. In order to make precise statements about which transitions fall in and out scope, this approach requires a timing specification language. Furthermore, when the designer wants to flatten a timing scope, the timing specifications must be reconciled in a consistent

manner—thus the operation demands a compositional logic.

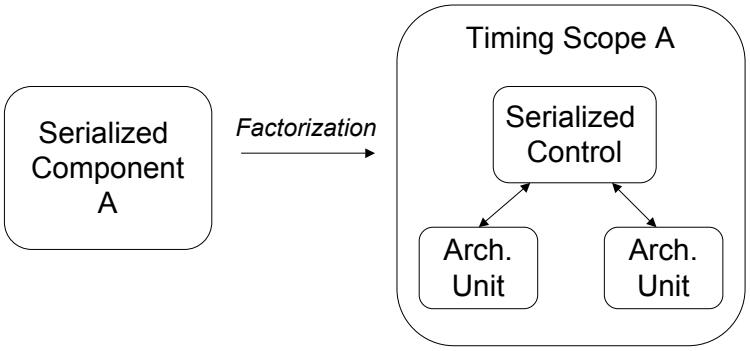


Figure 12: Defining transition visibility components with timing scopes.

Similarly, if all transitions are externally visible, what happens when one serializes a behavior table in a multi-component system? A timing change—e.g., an ALU that takes two steps instead of one step—could invalidate the timing expectations external components. This is a problem that Rath's Interface Specification Language (ISL [83], p. 18) begins to address, however Starfish does not annotate behavior tables with protocol specifications. Hypothetically, a transformation that changes the timeline would only be allowed in behavior tables with an interface protocol that tolerates changes. ISL, which specifies protocols with state machines, would have to be extended to define which protocol states allow timing changes. This view is largely equivalent to the timing scope view, except that scope flattening is mandatory; ISL would define the internal and external transitions, while permissibility of serialization constitutes the compositional logic.

Starfish does not attempt to solve this problem. Instead, it restricts factorization and serialization to avoid these timing mismatches. Factorization may follow serialization, but not the other way around. Serialization may only be applied to singleton

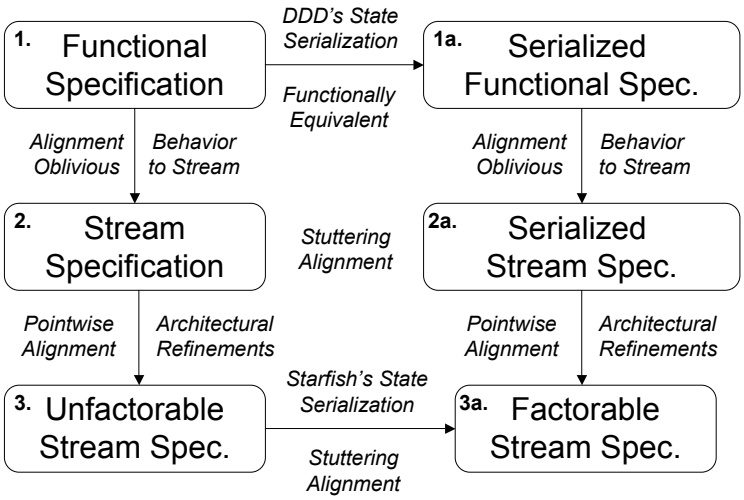


Figure 13: When DDD reaches a factorization impasse, the designer backtracks to the functional specification and introduces a serial control state; the workflow follows $1 \to 2 \to 3 \to 1 \to 1a \to 2a \to 3a$. In contrast, Starfish only operates on stream systems and directly serializes the "unfactorable" system, following the steps $2 \to 3 \to 3a$.

behavior table systems; in particular, this disallows serialization after factorization since factored systems contain multiple behavior tables. The designer must independently reconcile the shift from pointwise alignment to stuttering alignment with the system's intended use.

DDD's serialization facility saddles the designer with the same responsibility, though it conceals the alignment shift inside its behavioral algebra. When operator collisions prevent efficient factorizations in DDD, the designer can backtrack to the functional-behavioral representation and introduce a serial control state. After the behavioral manipulation, the previous derivation is re-applied to the serialized specification to produce a factorable system. Its result still contains the alignment shift.

Starfish applies serializations directly to stream-oriented systems without backtracking. Serialization tables show the register-architecture and intermediate values at the time of scheduling, improving the designer's ability to reason about the impact of scheduling possibilities when compared with DDD's relatively indirect process. Figure 13 summarizes the serialization workflow differences.

Definition 7.1. A *serialized behavior table* contains a distinguished column in the decision table, *ser*, which ranges over a prefix of the natural numbers. The table schema (Section 4.2.2, p. 63) for serialized behavior tables is:

$b : I \to O$	
P ser	S
$g_{\bar{\mathsf{n}}p}$	$t_{\bar{\mathsf{n}}s}$
0	$t_{\mathsf{n}s0}$
$g_{\mathsf{n}p}$ ⋮	⋮
M	$t_{\mathsf{n}sM}$

$\stackrel{\text{def}}{=}$

$b : I \to O$				
P	ser	c	c	S
$g_{\bar{\mathsf{n}}p}$		s_0	s_0	$t_{\bar{\mathsf{n}}s}$
$g_{\mathsf{n}p}$	#	s_0	s_1	$t_{\mathsf{n}s0}$
#	#	s_1	s_2	$t_{\mathsf{n}s1}$
⋮	⋮	⋮	⋮	⋮
#	#	s_{M-1}	s_M	$t_{\mathsf{n}s\mathsf{M}-1}$
#	#	s_M	s_0	$t_{\mathsf{n}s\mathsf{M}}$

(92)

where $\mathsf{s}_0, \ldots, \mathsf{s}_M$ is an enumerated type specified by the user. The schema expression, $g_{\mathsf{n}p}$, refers to a serialized guard, while $g_{\bar{\mathsf{n}}p}$ refers to the table's other guards. The schema shows that evaluation of a serialized guard with integers $0, \ldots, M$ in the ser column is defined by a control state c that steps through actions $0, \ldots, M$ with states $\mathsf{s}_0, \ldots, \mathsf{s}_M$. A *ser* column containing only # has no impact, and is a candidate for elimination in the same manner as other unspecified decision columns.

Example 7.2: This example translates the serialized stack calculator from Example 7.1 according to Definition 7.1. Let `s0, s1, s2, s3, s4` be an enumerated type with no functions or identities.

Inputs: (instr, a) Outputs:(res)						
(instr-cat instr)c		c:Seq.	res:Comb.	s:Seq.	x:Seq.	y:Seq.
psh-op	s0	s0	(top s)	(push s a)	#	#
pop-op	s0	s0	(top s)	(pop s)	#	#
alu-op	s0	s1	(top s)	s	(top s)	#
#	s1	s2	(top s)	(pop s)	x	#
#	s2	s3	(top s)	s	x	(top s)
#	s3	s4	(top s)	(pop s)	(alu instr x y)	#
#	s4	s0	(top s)	(push s x)	#	#

The semantics for this table match the streams in Example 7.1.

Definition 7.2 (Stuttering Map)**.** Let $B \in \{0,1\}^\infty$ be a bit-stream with infinitely many 1s, called a *stuttering mask*. The *stuttering output map* $B_{out} : A^\infty \to A^\infty$ induced by B is equal to $\lambda x.StutOut(x, B)$, where $Stut : \{0,1\}^\infty \times A^\infty \to A^\infty$ is defined co-recursively by:

$$StutOut(a, B) = \begin{array}{l} if(head(B) \\ \quad head(a) \\ \quad head(StutOut(tail(a), tail(B)))) \end{array} \; ! \; StutOut(tail(a), tail(B)) \qquad (93)$$

The stuttering output map behaves as a mask or characteristic-function for its input. The *stuttering input map* $B_{in} : A^\infty \to A^\infty$ induced by B is equal to $\lambda x.StutIn(x, head(x), B)$, where $StutIn : \{0,1\}^\infty \times A^\infty \to A^\infty$ is defined co-recursively by:

$$StutIn(a_0, a, B) = \begin{array}{l} if(head(B) \\ \quad a_0 \\ \quad head(a)) \end{array} \; ! \; \begin{array}{l} if(head(B) \\ \quad StutIn(head(a), tail(a), tail(B)) \\ \quad StutIn(a_0, a, tail(B))) \end{array} \qquad (94)$$

The stuttering input map stretches the input by the number of 0s.

Example 7.3: The stack calculator serialization in Example 7.1 introduces four serial transitions to the specification. The bit mask $B = (1, 1, 1, 0, 0, 0, 0, 1, \ldots)$ with the

maps B_{in} and B_{out} translate the inputs of the original system to serial-compatible inputs and the serial outputs to match the original outputs, respectively.

$inst$	$= (psh,$	$psh,$	$add,$					$psh,$	$\ldots)$
$a)$	$= (\ 5,$	$7,$	$\#,$					$25,$	$\ldots)$
B	$= (\ 1,$	$1,$	$1,$	0	0	0	0	$1,$	$\ldots)$
$B_{in}(inst)$	$= (psh,$	$psh,$	$add,$	$add,$	$add,$	$add,$	$add,$	$psh,$	$\ldots)$
$B_{in}(a)$	$= (\ 5,$	$7,$	$\#,$	$\#,$	$\#,$	$\#,$	$\#,$	$25,$	$\ldots)$

(95)

res	$= (0,$	$5,$	$7,$	$7,$	$5,$	$5,$	$0,$	$12,$	$\ldots)$
B	$= (1,$	$1,$	$1,$	0	0	0	0	$1,$	$\ldots)$
$B_{out}(res)$	$= (0,$	$5,$	$7,$					$12,$	$\ldots)$

(96)

Definition 7.3 (Correctness by Stuttering Alignment)**.** Let $S_1 : I^\infty \to O^\infty$ and $S_2 : I^\infty \to O^\infty$ be two stream systems. The S_2 is a stuttering simulation of S_1 if and only if for every $i \in I^\infty$ there is a stuttering mask B such that

$$Tr[S_1](I) = B_{out}(Tr[S_2](B_{in}(I))) \tag{97}$$

The serialization transformation produces a system that is a stuttering simulation of the original. A modified machine produces the mask for a given input stream. Simply add a boolean combinational signal B that is 0 on the guards that define the serial transitions and 1 otherwise; make B the only output stream. For example, the behavior table below produces a mask for the serialized stack calculator form Example 7.2:

Inputs: (instr, a) Outputs:(B)							
(instr-cat instr)	c	c:Seq.	res:Comb.	s:Seq.	x:Seq.	y:Seq.	B:comb
psh-op	s0	s0	(top s)	(push s a)	#	#	1
pop-op	s0	s0	(top s)	(pop s)	#	#	1
alu-op	s0	s1	(top s)	s	(top s)	#	0
#	s1	s2	(top s)	(pop s)	x	#	0
#	s2	s3	(top s)	s	x	(top s)	0
#	s3	s4	(top s)	(pop s)	(alu instr x y)	#	0
#	s4	s0	(top s)	(push s x)	#	#	1

(98)

Transformation 7 (Serialization)**.** *Let* $\mathsf{s}_0, \ldots, \mathsf{s}_{n-1}$ *be an enumerated type, tauS. Let* $InSer? : \tau_S \to Boolean$ *be* $\lambda x.\neg(x \in \{\mathsf{s}_1, \mathsf{s}_n\})$*. Let* $NextSer : \tau_S \to \tau_S$ *be* $\lambda \mathsf{s}_i.\mathsf{s}_{i+1(\mathbf{mod}\ n)}$*. Let* $EnterSer? : \tau_I \times \tau_X \to Boolean$*. Let* $F, G, H : \tau_I \times \tau_X \to \tau_X$*. For a given* $i \in \tau_I$*, let* $H_i(x) = H(i, x)$*. Suppose that*

$$EnterSer?(i, x) \Rightarrow F(i, x) = H_i^n(x) \tag{99}$$

H is a schedule for F at EnterSer?. Then

$$\begin{array}{l} SerSys(I) = Y \ \ where \\ \quad \begin{array}{rcl} B & = & sel(EnterSer?(I, X) \vee InSer?(S), 0, 1) \\ S & = & \mathsf{s}_0 \ ! \ sel(EnterSer?(I, X) \vee InSer?(S), NextSer(S), \mathsf{s}_0) \\ X & = & X_0 \ ! \ sel(EnterSer?(I, X) \vee InSer?(S), H(I, X), F(I, X)) \\ Y & = & G(I, X) \end{array} \end{array} \tag{100}$$

is a stuttering simulation for

$$\begin{array}{l} Sys(I) = Y \ \ where \\ \quad \begin{array}{rcl} X & = & X_0 \ ! \ F(I, X) \\ Y & = & G(I, X) \end{array} \end{array} \tag{101}$$

Moreover, signal B defines the stuttering mask for a given input stream I.

Proof. It suffices to show that the trace of state X is the same under the stuttering

mask; i.e., for all input streams I and states x:

$$Tr^*_X[Sys](I,x) = B_{out}(Tr^*_X[SerSys](B_{in}(I),x),\mathsf{s}_0) \tag{102}$$

Proof is by co-recursion on the input stream I. The head case is trivial:

$$\begin{array}{rcl} head(Tr^*_X[Sys](I,x)) & = & x \\ & = & head(B_{out}(Tr^*_X[SerSys](B_{in}(I),x,\mathsf{s}_0))) \end{array} \tag{103}$$

For the co-inductive step, let $=^*$ denote justification by the co-inductive hypothesis.

There are two cases for the tail: Suppose $EnterSer?(head(()i),x) = true$.

$$\begin{array}{ll} \multicolumn{2}{l}{tail(Tr^*_X[Sys](I,x))} \\ = & Tr^*_X[Sys](head(I),F(head(I),tail(x))) \\ = & Tr^*_X[SerSys](head(I),F(head(I),tail(x))) \\ = & Tr^*_X[SerSys](head(I),F(B_{in}(head(I)),tail(x))) \\ =^* & B_{out}(Tr^*_X[SerSys](head(I),sel(\ldots,(B_{in}(head(I)),tail(x))))) \\ = & tail(B_{out}(Tr^*_X[SerSys](I,x,\mathsf{s}_0))) \end{array} \tag{104}$$

Suppose $EnterSer?(head(()i),x) = false$.

$$\begin{array}{ll} \multicolumn{2}{l}{tail(Tr^*_X[Sys](I,x))} \\ = & Tr^*_X[Sys](head(I),F(head(I),tail(x))) \\ = & Tr^*_X[Sys](head(I),H^n_{head(I)}(tail(x))) \\ = & Tr^*_X[SerSys](head(I),H^n_{head(I)}(tail(x)),\mathsf{s}_0) \\ = & Tr^*_X[SerSys](head^n(B_{in}(I)),H^n_{head(I)}(tail(x)),\mathsf{s}_0) \\ =^* & tail(B_{out}(H(head(B_{in}(I)),tail(x))\ ! \qquad\qquad)) \\ & \qquad H(head^2(B_{in}(I)),H(head(B_{in}(I)),tail(x)))\ !\ \ldots \\ & \qquad !\ Tr^*_X[SerSys](head^n(B_{in}(I)),H^n_{head(I)}(tail(x)),\mathsf{s}_0) \\ = & tail(B_{out}(Tr^*_X[SerSys](B_{in}(I),x,\mathsf{s}_0))) \end{array} \tag{105}$$

The horizontal dots indicate facts that are proven by induction on n. The crucial

observations are that B_{out} delays its output for n steps and that B_{in} holds its input for n steps so that $head^i(B_{in}(I))$=$head(I)$ for $1 \leq i \leq n$. This is due to the n linear control states defined by S. □

7.2 How Starfish supports serialization

Starfish introduces serialization tables to help the designer produce sequence of table terms whose cumulative sequential evaluation equals the serialization target. In terms of the stream system transformation statement, Starfish supports the construction of H such that $H_i^n(x) = F(i, x)$ for an n-step serialization. Three additional facilities make scheduling a more tractable task: serialization tables, retiming within a serial flow, and equational logic within a serialization table's evaluation display.

Like behavior tables, serialization tables are split into left and right halves. The left side, or serial action table, displays sequential update expressions for each signal. Since control flow is linear, no decision table is necessary. The right side contains the evaluation table. Its cells contain the symbolic evaluation of each signal after execution of the corresponding action in the serial action table. The very last row contains target expressions; i.e., the nested action terms that are the subject of serialization. In table scheme notation, the serialization table for the actions $f(S)$ is:

S	S
$h_0(S)$	$h_0(S)$
$\vdots$	$\vdots$
$h_i(S)$	$h_i \circ \ldots \circ h_0(S)$
$\#$	$f(S)$

(106)

The left-hand-side is called *scheduling table*. The right-hand side is called the *evaluation table*; it shows the cumulative symbolic evaluation of terms in the scheduling table. The bottom row is the *evaluation requirement*. A schedule is valid when he evaluation table's last row equals the evaluation requirement.

Initial serialization

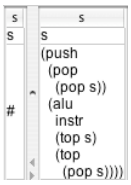
Serialization begins by specifying a subject action and a subset of sequential signals. The `begin-serialization` command creates an initial serialization table with one row. Both scheduling and evaluation tables have the chosen sequential signals as their column headings. The initial values for the actions are chosen at this time. The designer manipulates the serialization tables with the following commands:

- `insert-col` adds a new sequential signal into the table; the evaluation requirement is `#:<type>`.
- `insert-row` adds a step to the schedule beginning at a specified row and displacing existing steps downward.
- `new-ser-row` appends a step to the schedule.
- `set-cell` overwrites the scheduling term at a specified signal and step.
- `ser-eval-ident` applies an algebraic identity to a term in the evaluation table.
- `oblige-ser-eval-ident` applies an arbitrary rewrite to a term in the evaluation table. This rewrite becomes a condition for the derivation's soundness, and must be externally validated.

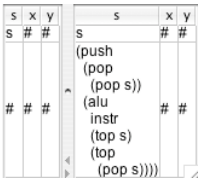
After the designer creates a valid schedule, the command `insert-ser-tab` incorporates it into the behavior table using the `ser`-column decision table notation from Definition 7.1.

Example 7.4: This example shows how serialization tables facilitate schedule specification in Example 7.2's stack calculator. The serialization process in Starfish begins by specifying the target action via the corresponding guards (`alu-op`).

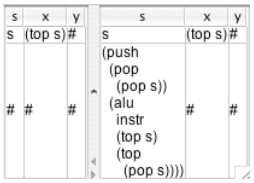
The initial serialization table holds the one sequential signal `s`. Since `s` is a subterm of the serialization subject `(push (alu inst ...) (pop ...))`, the default initial action holds `s`'s value.



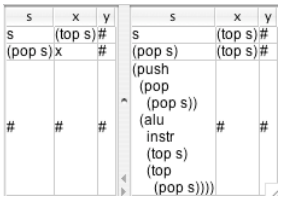
The next two steps introduce two sequential signals for holding intermediate values, `x` and `y`. They behave as data buffers for the `alu` operation.



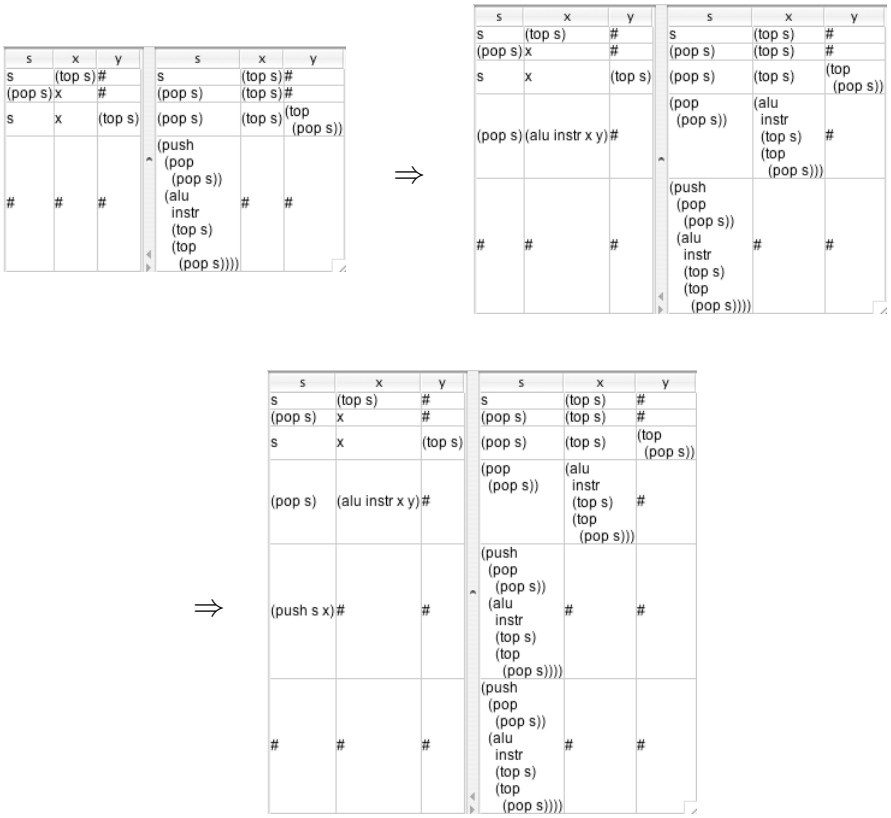
The first serialization step stores `(top s)` in the `x`.



When declaring new actions, the default is to hold values for "live" registers and update `#` signals with `#`. Specifying the example's next action only requires declaration that `(pop s)` updates `s`. Updates for `x` and `y` are automatically generated. This is also the first step where evaluation table differs from the serial action table.



The following three tables declare the next three serial actions. The bottom row displays the evaluation goals for each signal.



Rescheduling

In addition to serialization of actions, Starfish supports rescheduling or reserialization within a subsequence of scheduled actions. The `resume-serialization` creates a serialization table for contiguous set of serial actions in a behavior table. The symbolic evaluation of this sub-schedule form the new evaluation requirements. After the designer produces a valid rescheduling, the `insert-ser-tab` commits the new sub-schedule into the behavior table. If the new result has a different number of steps, the enclosing schedule is expanded or contracted to accommodate new size.

Rescheduling does not derive any system that could not be derived with serialization alone. Why not simply undo the original serialization and specify the new

one? There are many cases where this strategy defies the principle of informed and intelligent designer interaction. For instance, type translation of abstract pairs into a register-heap representation can produce very large terms. A refine-data-then-serialize strategy can lead to term explosion, pushing the behavior tables beyond their utility as perspicuous system representations. However, the serialize-refine-data-then-reschedule strategy exchanges term size for larger tables. In particular the `resume-serialization` facility limits the size of target terms by only rescheduling a *portion* of a schedule.

Example 7.5: The following behavior table implements stacks with a heap: stack pointer `s*`, memory signal `mem`, and heap horizon pointer `ptr`. Full details of the translation specification appear in next chapter's Figure 15. The table below shows the translation applied to the serialized stack calculator from Example 7.1.

Inputs: (instr, a)
Outputs: (res)

ser	(inst-cat instr)	res:Comb.	s*:Seq.	mem:Seq.	ptr:Seq.	x:Seq.	y:Seq.
#	psh-op	(top s)	ptr	(wr mem ptr [a s*])	(inc ptr)	#	#
#	drp-op	(top s)	(2nd (rd mem s*))	mem	ptr	#	#
0	alu-op	(top s)	s*	mem	ptr	(1st (rd mem s*))	#
1	alu-op	(top s)	(2nd (rd mem s*))	mem	ptr	x	#
2	alu-op	(top s)	s*	mem	ptr	x	(1st (rd mem s*))
3	alu-op	(top s)	(2nd (rd mem s*))	mem	ptr	(alu instr x y)	#
4	alu-op	(top s)	ptr	(wr mem ptr [x s*])	(inc ptr)	#	#

Since the heap stores cells containing stack value and pointer to a stack's tail, this representation calculates the *top* and *pop* in one memory access. Rescheduling optimizes the two read accesses of steps 1 and 2 to a single read access as follows:

Rescheduling begins guard and subrange specification; in this case `alu-op` and steps `1` through `2`. The evaluation requirements show the sequential evaluation of these two steps for each signal.

s*	mem	ptr	x	y
s*	mem	ptr	#	#
#	#	#	#	#

s*	mem	ptr	x	y
s*	mem	ptr	#	#
(2nd (rd mem s*))	mem	ptr	(1st (rd mem s*))	#

A new signal `cell:[ind integer]` holds the result of a heap read. The *top* and *pop* values are fields of this tuple.

s*	mem	ptr	x	y	cell
s*	mem	ptr	#	#	(rd mem s*)
#	#	#	#	#	#

s*	mem	ptr	x	y	cell
s*	mem	ptr	#	#	(rd mem s*)
(2nd (rd mem s*))	mem	ptr	(1st (rd mem s*))	#	#

The schedule's second step transfers the fields of `cell` to registers `s*` and `x`, satisfying the evaluation requirements.

s*	mem	ptr	x	y	cell
s*	mem	ptr	#	#	(rd mem s*)
(2nd cell)	mem	ptr	(1st cell)	#	#
#	#	#	#	#	#

s*	mem	ptr	x	y	cell
s*	mem	ptr	#	#	(rd mem s*)
(2nd (rd mem s*))	mem	ptr	(1st (rd mem s*))	#	#
(2nd (rd mem s*))	mem	ptr	(1st (rd mem s*))	#	#

`insert-ser-tab` commits the new sub-schedule to the behavior table.

Inputs: (instr, a)
Outputs: (res)

ser	(inst-cat instr)	res:Comb.	s*:Seq.	mem:Seq.	ptr:Seq.	x:Seq.	y:Seq.	cell:Seq.
#	psh-op	(top s)	ptr	(wr mem ptr [a s*])	(inc ptr)	#	#	#
#	drp-op	(top s)	(2nd (rd mem s*))	mem	ptr	#	#	#
0	alu-op	(top s)	s*	mem	ptr	#	#	(rd mem s*)
1	alu-op	(top s)	(2nd cell)	mem	ptr	(1st cell)	#	#
2	alu-op	(top s)	s*	mem	ptr	x	(1st (rd mem s*))	#
3	alu-op	(top s)	(2nd (rd mem s*))	mem	ptr	(alu instr x y)	#	#
4	alu-op	(top s)	ptr	(wr mem ptr [x s*])	(inc ptr)	#	#	#

This table shows the result of rescheduling steps 2 and 3 in a similar way. The new schedule takes one more step, since the ALU must wait for the transfer of the *top* out of `cell` and into `x`.

Inputs: (instr, a)
Outputs: (res)

ser	(inst-cat instr)	res:Comb.	s*:Seq.	mem:Seq.	ptr:Seq.	x:Seq.	y:Seq.	cell:Seq.
#	psh-op	(top s)	ptr	(wr mem ptr [a s*])	(inc ptr)	#	#	#
#	drp-op	(top s)	(2nd (rd mem s*))	mem	ptr	#	#	#
0	alu-op	(top s)	s*	mem	ptr	#	#	(rd mem s*)
1	alu-op	(top s)	(2nd cell)	mem	ptr	(1st cell)	#	#
2	alu-op	(top s)	s*	mem	ptr	x	#	(rd mem s*)
3	alu-op	(top s)	(2nd cell)	mem	ptr	x	(1st cell)	#
4	alu-op	(top s)	s*	mem	ptr	(alu instr x y)	#	#
5	alu-op	(top s)	ptr	(wr mem ptr [x s*])	(inc ptr)	#	#	#

Starfish only applies the schedule to a behavior table when the last row of the evaluation table is syntactically equal the evaluation targets. If the terms are equal through sequence of algebraic rewrites, the commitment is still valid. While Starfish is does not determine equivalence with a rewriting algorithm, the designer can "argue" equivalence by applying algebraic identities to the evaluation table. Thus, `insert-ser-tab` can incorporate schedules which produce outcomes that are semantically equivalent to the evaluation requirements.

Example 7.6: Suppose that the memory of cells from Example 7.5 supports componentwise writing, so that the first and second fields of the memory cell may be written separately; i.e.,

$$wr_1(wr_2(m, i, f_2), i, f_1) = wr(m, i, [f_1\ f_2]) \tag{107}$$

Then a further reserialization may write the stack cell in as the tail address and new datum become available. This new schedule is proposed with the following

reserialization of steps 4 and 5:

s*	mem	ptr	x	y	cell
#	(wr2 mem ptr s*)	ptr	(alu instr x y)	#	#
ptr	(wr1 mem ptr x)	(inc ptr)	#	#	#
#	#	#	#	#	#

s*	mem	ptr	x	y	cell
#	(wr2 mem ptr s*)	ptr	(alu instr x y)	#	#
ptr	(wr1 (wr2 mem ptr s*) ptr (alu instr x y))	(inc ptr)	#	#	#
ptr	(wr mem ptr [(alu instr x y) s*])	(inc ptr)	#	#	#

Evaluating the schedule produces a different, but semantically equivalent outcome for signal `mem`. Applying (107) rewrites the expression as follows:

```
(wr1                          (wr
  (wr2 mem ptr s*)      ⇒       mem
  ptr                           ptr
  (alu instr x y))              [(alu instr x y)
                                 s*])
```

The outcome matches the evaluation requirements for `mem`, validating the schedule.

s*	mem	ptr	x	y	cell
#	(wr2 mem ptr s*)	ptr	(alu instr x y)	#	#
ptr	(wr1 mem ptr x)	(inc ptr)	#	#	#
#	#	#	#	#	#

s*	mem	ptr	x	y	cell
#	(wr2 mem ptr s*)	ptr	(alu instr x y)	#	#
ptr	(wr mem ptr [(alu instr x y) s*])	(inc ptr)	#	#	#
ptr	(wr mem ptr [(alu instr x y) s*])	(inc ptr)	#	#	#

Inserting the schedule into the behavior table completes the example:

Inputs: (instr, a)
Outputs: (res)

ser	(inst-cat instr)	s*:Seq.	mem:Seq.	ptr:Seq.	res:Comb.	x:Seq.	y:Seq.	cell:Seq.
#	psh-op	ptr	(wr mem ptr [a s*])	(inc ptr)	(top s)	#	#	#
#	drp-op	(2nd (rd mem s*))	mem	ptr	(top s)	#	#	#
0	alu-op	s*	mem	ptr	(top s)	#	#	(rd mem s*)
1	alu-op	(2nd cell)	mem	ptr	(top s)	(1st cell)	#	#
2	alu-op	s*	mem	ptr	(top s)	x	#	(rd mem s*)
3	alu-op	(2nd cell)	mem	ptr	(top s)	x	(1st cell)	#
4	alu-op	#	(wr2 mem ptr s*)	ptr	(top s)	(alu instr x y)	#	#
5	alu-op	ptr	(wr1 mem ptr x)	(inc ptr)	(top s)	#	#	#

The evaluation table rewrite could have been avoided by applying the identity to the target term prior to reserialization. In theory this is always possible, however

it is more convenient to justify the rewrites as the schedule develops, rather than anticipate all necessary rewriting in advance.

Chapter 8

Data Refinement

Data representation has deep consequences for the space of attainable architectures. Since behavioral specifications frequently operate on abstract data types, a mechanism which allows selective implementation of datatypes is critical for the top end of top-down design methodologies. Starfish implements a *data refinement* facility which expresses homomorphisms from abstract types to implementation types with extensions. Refinement falls into two broad categories: when the refinement homomorphism is one-to-one, and when the refinement homomorphism admits multiple representations of the same abstract object. This chapter presents definitions and refinement transformations for one-to-one homomorphisms, one-to-many refinements, and *stateful refinements*. Abstract stacks—a time-honored vehicle [46] for presenting refinement procedures—illustrate the software support for refinement. The chapter concludes with an extended example that illustrates how data refinement, serialization, and factorization interact to impose a non-trivial high level architecture on a behavior oriented specification.

8.1 One-to-one refinements

Data refinement changes system behavior. Consider the abstract stack signature presented in Section 3.2.2. Common stack implementations represent stacks with an array and a pointer to the top of the stack. In addition to a finer grained view of the data representation, data refinement also exposes a finer grained view of data operations. Now `push` is a multistep operation, first writing the value to the array and then incrementing the horizon pointer. The approach developed here expresses this duality by modifying term expressions, rather than term expressions *and* behavior. Consequent control flow modifications are accessible via serialization.

The following theorem shows how to change a sequential equation over an abstract type into an equivalent pair of equations - a sequential equation operating on an implementation type and a combinational equation that recovers or *coerces* the abstract type from the implementation signal. Functions ρ and α are refinement and abstraction *coercions*.

Theorem 8.1 (One-to-one Refinement)**.** *Let A and R be two sorts with functions $\rho : A \to R$ and $\alpha : R \to A$ such that for all $a \in A$, $\alpha(\rho(a)) = a$. Assume that there is a consistent system of term-level identities such that $\rho(T(a, i, s, c)) = T_\rho(\rho(a), i, s, c)$. Then the following systems are pointwise equivalent:*

$$\begin{array}{lll} \vdots & \vdots & \vdots \\ X & = x_0 \quad ! & T(X', I, S, C) \\ \vdots & \vdots & \vdots \end{array} \Leftrightarrow \begin{array}{lll} \vdots & \vdots & \vdots \\ X & = & \alpha(X') \\ X' & = \rho(x_0) \quad ! & T_\rho(X', I, S, C) \\ \vdots & \vdots & \vdots \end{array} \tag{108}$$

Proof. This proof argues only by pointwise equivalent transformations to ensure that the two systems are pointwise equivalent. First introduce a combinational signal X':

$$\begin{array}{lll} X & = x_0 \quad ! & T(X, I, S, C) \\ X' & = & \rho(X) \end{array} \tag{109}$$

Expand the identity function on A into $\alpha \circ \rho$:

$$\begin{array}{lll} X & = \alpha(\rho(x_0)) \quad ! & \alpha(\rho(T(X, I, S, C))) \\ X' & = & \rho(X) \end{array} \tag{110}$$

Rewrite $\rho(T(X, I, S, C)) = T_\rho(\rho(X), I, S, C)$.

$$\begin{array}{lll} X & = \alpha(\rho(x_0)) \quad ! & \alpha(T_\rho(\rho(X), I, S, C)) \\ X' & = & \rho(X) \end{array} \tag{111}$$

Perform a combinational identification, replacing $\rho(X)$ with X':

$$\begin{array}{lll} X & = \alpha(\rho(x_0)) \quad ! & \alpha(T_\rho(X', I, S, C)) \\ X' & = & \rho(X) \end{array} \tag{112}$$

Convert X' to a sequential signal (Transformation 6, p. 53):

$$\begin{array}{lll} X & = \alpha(\rho(x_0)) \quad ! & \alpha(T_\rho(X', I, S, C)) \\ X' & = \rho(\alpha(\rho(x_0))) \quad ! & \rho(\alpha(T_\rho(X', I, S, C))) \end{array} \tag{113}$$

Reduce X' according to $\rho(\alpha(r)) = r$ when r is a valid representation value in R; i.e., $r = \rho(a)$ for some $a \in A$ or $r = f(r')$ for some valid representation value $r' \in R$ and implementation function $f : R -> R$ (given a valid representation f produces a valid representation).

$$\begin{array}{lll} X & = \alpha(\rho(x_0)) \quad ! & \alpha(T_\rho(X', I, S, C)) \\ X' & = \rho(x_0) \quad ! & T_\rho(X', I, S, C) \end{array} \tag{114}$$

Convert X to a combinational signal based on the definition of X'.

$$\begin{array}{lll} X & = & \alpha(X') \\ X' & = \rho(x_0)\ ! & T_\rho(X', I, S, C) \end{array} \tag{115}$$

□

This theorem requires a system of identities which can transform a the refinement of a function applied to an abstract, $\rho(f(a)$, into an implementation function applied to the refinement of an abstract type, $f_\rho(\rho(a))$. Refinement declarations in Starfish capture these assertions. Refinements declare type signatures for ρ and α, as well as a system of homomorphic identities for functions with domain or range in the abstract type. A complete set of homomorphic identities demonstrates the action of ρ on all constants of the abstract type, all functions that consume the abstract type, and all functions that produce the abstract type.

Sort label:	ind	$memory$
Signature Kind:	Standard	Parameterized over $\{addr\ data\}$
Constants:	0_{ind}	$m_0 : memory\{addr\ data\}$
Functions:	$inc : ind \rightarrow ind$ $dec : ind \rightarrow ind$	$wr : memory\{addr\ data\} \times addr \times data \rightarrow memory\{addr\ data\}$ $rd : memory\{addr\ data\} \times addr \rightarrow data$
ID Variables:	$i : ind$	$m : memory\{addr\ data\}$ $a : addr$ $d, e : data$
Identities:	$dec(inc(i)) = i$ $sel(eq?(0_{ind}, i),$ $\quad i, inc(dec(i))) = i$	$rd(wr(m, a, d), a) = d$ $wr(m, a, rd(m, a)) = m$ $wr(wr(m, a, d), a, e) = wr(m, a, e)$

Figure 14: The *ind* signature instantiates the address parameter of the memory signature for our examples.

Example 8.1: Consider a parameterized type declaration for *memory* and the declaration for *ind* in Figure 14. These signatures will implement the "array with pointer"

refinement of abstract stacks of integers. The following diagrams express the homomorphism ρ that embeds abstract stacks into a tuple of an indexed memory and index. Let s be an abstract stack, d be a stack value, $m = \pi_0(\rho(s))$ and $i = \pi_1(\rho(s))$.

$$\begin{array}{ccc} S & \xrightarrow{push(s,d)} & S \\ {\scriptstyle\rho}\downarrow & & \downarrow{\scriptstyle\rho} \\ M \times I & \xrightarrow[[wr(m,i,d)\ \ inc(i)]]{} & M \times I \end{array}$$

$$\begin{array}{ccccccc} S & \xrightarrow{pop(s)} & S & \quad S & \xrightarrow{top(s)} & \mathbb{Z} \\ {\scriptstyle\rho}\downarrow & & \downarrow{\scriptstyle\rho} & {\scriptstyle\rho}\downarrow & & \| \\ M \times I & \xrightarrow[[wr(m,dec(i),0)\ \ dec(i)]]{} & M \times I & M \times I & \xrightarrow[rd(m,dec(i))]{} & \mathbb{Z} \end{array} \tag{116}$$

The diagrams show how the refinement type implements functions from the abstract type, but conventional equational expressions better capture the identities' role as rewrite rules:

$$\begin{array}{lcl} \rho(push(s,d)) & = & [wr(\pi_0(\rho(s)), \pi_1(\rho(s)), d)\ inc(\pi_1(\rho(s)))] \\ \rho(pop(s)) & = & [wr(\pi_0(\rho(s)), dec(\pi_1(\rho(s))), 0)\ dec(\pi_1(\rho(s)))] \\ top(s) & = & rd(\pi_0(\rho(s)), dec(\pi_1(\rho(s)))) \end{array} \tag{117}$$

The refinement procedure combines sequential identification (Transformation 8.1) followed by homomorphic rewrites. The system below shows the refinement theorem applied to signal s in the stack calculator from Example 3.17.

$$
\begin{array}{l}
StackCalc(instr, a) = res \\
\quad where \\
\quad\quad s^* \;=\; \rho(push(mt,0))\ ! \\
\quad\quad\quad\quad\quad \rho(sel(instCat(instr), push(s,a), pop(s), \\
\quad\quad\quad\quad\quad\quad\quad push(pop(pop(s)), alu(instr, top(s), top(pop(s))))))) \\
\quad\quad s \;=\; \alpha(s^*) \\
\quad\quad res \;=\; top(s)
\end{array}
\tag{118}
$$

The following sequence of rewrites show how the refinement identities above and the rewrite rule $s^* = \rho(s)$ from Transformation 8.1 change abstract stacks into memories and pointers.

$$
\begin{array}{lll}
1. & \rho(push(mt,0)) & = [wr(\pi_0(\rho(mt)), \pi_1(\rho(mt)), 0) inc(\pi_1(\rho(mt)))] \\
2. & & = [wr(\pi_0([m_0, 0_{ind}]), \pi_1([m_0, 0_{ind}]), 0)\ inc(\pi_1([m_0, 0_{ind}]))] \\
3. & & = [wr(m_0, 0_{ind}, 0)\ inc(0_{ind})] \\
4. & \rho(sel(instCat(instr), push(s,a), pop(s), push(pop(\ldots)))) & \\
 & & = sel(instCat(instr), \rho(push(s,a)), \rho(pop(s)), \rho(push(pop(\ldots)))) \\
5. & \rho(push(s,a)) & = [wr(\pi_0(\rho(s)), \pi_1(\rho(s)), a)\ inc(\pi_1(\rho(s)))] \\
6. & & = [wr(\pi_0(s^*), inc(\pi_1(s^*)), a)\ inc(\pi_1(s^*))] \\
7. & \rho(pop(s)) & = [wr(\pi_0(\rho(s)), dec(\pi_1(\rho(s))), 0)\ dec(\pi_1(\rho(s)))] \\
8. & & = [wr(\pi_0(s^*), dec(\pi_1(s^*)), 0)\ dec(\pi_1(s^*))]
\end{array}
$$

Line 1 begins the rewrite of s^*'s initial value by applying the *push* identity. Line 2 rewrites mt as the zero-memory m_0 and zero pointer 0_{ind}. Line 3 applies the projectors to the tuple, producing a simplified expression. Line 4 continues by applying a selector identity to the sequential update term: functions distribute over the branches of a selector. Line 5's rewrite focuses on the selector's *push* branch by applying the corresponding homomorphic identity. Line 6 eliminates the refinement operator by replacing $\rho(s)$ with s^*. Similarly Line 7 rewrites the *pop* branch with the *pop* identity, and Line 8 replaces $\rho(s)$ with s^*.

Adding the combinational equations

$$\begin{array}{rcl} m & = & \pi_0(s^*) \\ i & = & \pi_1(s^*) \end{array} \tag{119}$$

to the system enables further simplification of the rewritten terms, resulting in more intuitive expressions: e.g., $\rho(pop(s)) = [wr(m, dec(i), 0)\ dec(i)]$ In this context, we continue the rewrite of the final selector branch:

9. $\rho(push(pop(pop(s)), ALUinst(top(s), top(pop(s)))))$
10. $= [wr(\pi_0(\rho(pop(pop(s)))), \pi_1(\rho(pop(pop(s)))), ALUinst(top(s), top(pop(s))))$
$inc(\pi_1(\rho(pop(pop(s)))))]$
11. $= [wr(\pi_0([wr(\pi_0(\rho(pop(s))), dec(\pi_1(\rho(pop(s)))), 0)\ dec(\pi_1(\rho(pop(s))))]),$
$\pi_1([wr(\pi_0(\rho(pop(s))), dec(\pi_1(\rho(pop(s)))), 0)\ dec(\pi_1(\rho(pop(s))))]),$
$ALUinst(top(s), top(pop(s))))$
$inc(\pi_1([wr(\pi_0(\rho(pop(s))), dec(\pi_1(\rho(pop(s)))), 0)\ dec(\pi_1(\rho(pop(s))))]))]$
12. $= [wr(wr(\pi_0(\rho(pop(s))), dec(\pi_1(\rho(pop(s)))), 0), dec(\pi_1(\rho(pop(s)))),$
$ALUinst(top(s), top(pop(s))))$
$inc(dec(\pi_1(\rho(pop(s)))))]$
13. $= [wr(wr(\pi_0([wr(\pi_0(s^*), dec(\pi_1(s^*)), 0)\ dec(\pi_1(s^*))]),$
$dec(\pi_1([wr(\pi_0(s^*), dec(\pi_1(s^*)), 0)\ dec(\pi_1(s^*))])), 0),$
$dec(\pi_1([wr(\pi_0(s^*), dec(\pi_1(s^*)), 0)\ dec(\pi_1(s^*))])),$
$ALUinst(top(s), top(pop(s))))$
$inc(dec(\pi_1([wr(\pi_0(s^*), dec(\pi_1(s^*)), 0)\ dec(\pi_1(s^*))]))))]$
14. $= [wr(wr(wr(\pi_0(s^*), dec(\pi_1(s^*)), 0), dec(dec(\pi_1(s^*))), 0), dec(dec(\pi_1(s^*))),$
$ALUinst(top(s), top(pop(s))))$
$inc(dec(dec(\pi_1(s^*))))]$
15. $= [wr(wr(wr(m, dec(i), 0), dec(dec(i)), 0), dec(dec(i)),$
$ALUinst(top(s), top(pop(s))))$
$inc(dec(dec(i)))]$
16. $= [wr(wr(m, dec(i), 0), dec(dec(i)), ALUinst(top(s), top(pop(s))))\ dec(i)]$

In the above sequence, Lines 9-14 are rote applications of the homomorphic identities and tuple projection identities: Line 10 expands the homomorphic identity for *push*, Line 11 expands the homomorphic identity for *pop*, Line 12 simplifies projections of

tuples, Line 13 expands the homomorphic identity for *pop*, Line 14 simplifies projections of tuples. Line 15 replaces instances of $\rho(s)$ with s^*. Line 16 exploits two identities from the *ind* and *memory* declarations (Figure 14): $inc(dec(i))) = i$, and $wr(wr(m, a, d), a, e) = wr(m, a, e)$.

The remaining stack references are a result of *top* accesses. These terms do not have a coercion operator, however the homomorphic identity,

$$top(s) = rd(\pi_0(\rho(s)), dec(\pi_1(\rho(s)))) \tag{120}$$

does not require one.

17. $ALUinst(top(s), top(pop(s)))$
18. $= ALUinst(rd(\pi_0(\rho(s)), dec(\pi_1(\rho(s)))), rd(\pi_0(\rho(pop(s))), dec(\pi_1(\rho(pop(s))))))$
19. $= ALUinst(rd(\pi_0(s^*), dec(\pi_1(s^*))),$
$rd(\pi_0([wr(\pi_0(s^*), dec(\pi_1(s^*)), 0)\ dec(\pi_1(s^*))]),$
$dec(\pi_1([wr(\pi_0(s^*), dec(\pi_1(s^*)), 0)\ dec(\pi_1(s^*))])))$
20. $= ALUinst(rd(\pi_0(s^*), dec(\pi_1(s^*))),$
$rd(wr(\pi_0(s^*), dec(\pi_1(s^*)), 0), dec(dec(\pi_1(s^*)))))$
21. $= ALUinst(rd(m, dec(i)), rd(wr(m, dec(i), 0), dec(dec(i))))$

Lines 17-20 follow the same homomorphic identity, projection identity, replace-$\rho(s)$, recurse pattern of the previous rewritings. Line 21 simplifies the terms with the combinational signals m and i, as before.

The following identities can further reduce the term in Line 21. Introduce a index equality predicate *eq*? such that

$$\begin{aligned} rd(wr(m, a, d), b) &= sel(eq?(a, b), d, rd(m, b)) \\ eq?(i, dec(i)) &= false \end{aligned} \tag{121}$$

These identities require external validation, but are clearly true for a number of plausible models; e.g., *ind* as the integers, and *memory* as an association list of integers keyed by integers.

$$
\begin{aligned}
21.\quad &= ALUinst(rd(m, dec(i)), rd(wr(m, dec(i), 0), dec(dec(i))))\\
22.\quad &= ALUinst(rd(m, dec(i)), sel(eq?(dec(i), dec(dec(i))), 0, rd(m, dec(dec(i)))))\\
23.\quad &= ALUinst(rd(m, dec(i)), sel(false, 0, rd(m, dec(dec(i)))))\\
24.\quad &= ALUinst(rd(m, dec(i)), rd(m, dec(dec(i))))
\end{aligned}
$$

The following system is the result of all these rewriting steps:

$$
\begin{aligned}
&StackCalc(instr, a) = res\\
&\quad where\\
&\quad\quad s^* = [wr(m_0, 0_{ind}, 0)\ inc(0_{ind})]\ !\\
&\quad\quad\quad\quad sel(instCat(instr),\\
&\quad\quad\quad\quad\quad\quad [wr(m, i, a)\ inc(i)]\\
&\quad\quad\quad\quad\quad\quad [wr(m, dec(i), 0)\ dec(i)]\\
&\quad\quad\quad\quad\quad\quad [wr(wr(m, dec(i), 0), dec(dec(i)),\\
&\quad\quad\quad\quad\quad\quad\quad\quad ALUinst(rd(m, dec(i)), rd(m, dec(dec(i)))))\ dec(i)]\\
&\quad\quad res = rd(m, dec(i))\\
&\quad\quad m = \pi_0(s^*)\\
&\quad\quad i = \pi_1(s^*)
\end{aligned}
\tag{122}
$$

Note that the final system safely eliminates the equation $s = \alpha(s^*)$ since no term reference to s exists.

This example shows how data refinement in stream systems combines the sequential signal translation (Transformation 8.1) and term rewriting according to the refinement homomorphism. The rewriting steps are rote and tedious, therefore easily automated. Starfish automates the application of the homomorphic identities, projector identities, and $\rho(s)$ replacements.

8.2 One-to-many refinements

Most interesting data refinement schemes do not uniquely represent an abstract type. For instance, the previous section's *pop* representation not only decrements the stack pointer, but *overwrites* the memory with the initial value 0. The common implementation of stacks simply decrements the pointer. This complicates the formalization, because there are now arbitrarily many array-pointer pairs that represent a particular stack; i.e., there may be arbitrary junk in the array beginning at the horizon pointer. Thus the coercion operator ρ cannot be a function from the abstract type to the refinement type.

A slight modification of the homomorphism rules yields a workable solution. Starfish adds another parameter to multi-representation refinements that specifies the particular representation.

Definition 8.1. A *multi-representation refinement function* from A to R, $\rho : A \times R \to R$ and its *abstraction function* $\alpha : R \to A$ satisfy the following identities, for all $r \in R$, $a \in A$, $f_A : A \to A$, $g_A : X \to A$, and $h_A : A \to X$:

$$\alpha(\rho(f_A(a), r)) = \alpha(f_R(\rho(a, \rho(a, r)))) \tag{123}$$

$$\alpha(\rho(g_A(x), r)) = \alpha(g_R(x)) \tag{124}$$

$$h_A(a) = h_R(\rho(a, r)) \tag{125}$$

$$\alpha(\rho(a, r)) = a \tag{126}$$

where $f_R : R \to R$, $g_R : X \to R$, and $h_R : R \to X$ are functions in the implementation type.

Example 8.2: The conventional stack implementation that only decrements the stack pointer on *pop* is a multi-representation refinement. Let $\rho : Stack\{data\} \times [mem\{ind\ data\}\ ind] \to [mem\{ind\ data\}\ ind]$ be defined by:

$$\rho(s, [m\ i]) = [wr(wr(\ldots wr(m, 0, top(s))), inc^{len-1}(0_{ind}), top(pop^{len-1}(0_{ind})))\ len(s)] \tag{127}$$

Similarly, let the abstraction function be defined by $\alpha : [ind\ mem\{ind\ data\}] \to Stack\{data\}$

$$\alpha([m\ i]) = push(push(\ldots push(mt, rd(m, 0_{ind}))\ldots), rd(m, inc^{len-1}(0_{ind}))) \tag{128}$$

The homomorphic identities corresponding to requirements (123), (124), and (125) are:

$$\begin{array}{lcl} \rho(mt, [m\ i]) & = & [m\ 0_{ind}] \\ \rho(push(s, a), [m\ i]) & = & [wr(m^*, i^*, a)\ inc(i^*)] \\ & & \quad where\ [m^*\ i^*] = \rho(s, [m\ i]) \\ \rho(pop(s), [m\ i]) & = & [m^*\ dec(i^*)] \\ & & \quad where\ [m^*\ i^*] = \rho(s, [m\ i]) \\ \rho(top(s), [m\ i]) & = & rd(m*, dec(i^*)) \\ & & \quad where\ [m^*\ i^*] = \rho(s, [m\ i]) \end{array} \tag{129}$$

A one-to-many refinement provides a choice of representations for each abstract

value. Shifting from one representation to another produces an equivalent system under pointwise alignment. For instance, in the stack representation, the "junk" values after the pointer have no impact on abstraction operator. More interestingly, garbage collection on a heap trades one representation for a more compact representation of the same abstract type; e.g., lists or stacks. The following transformation formalizes the equivalence of stream systems under a re-embedding of the abstract type into the representation type.

Transformation 8 (Re-embedding)**.** *Let $\rho : A \times R \to R$ and $\alpha : R \to A$ be multi-representation coercion operators. The following are equivalent systems by pointwise alignment for all choices of $I_R \in R^\infty$:*

$$\begin{array}{l} Sys_0(I) = O \text{ where} \\ \quad \begin{array}{lcl} \vdots & = & \vdots \\ X & = & \alpha(X_R) \\ X_R & = & x_R \mathbin{!} F(X_R, I) \\ \vdots & = & \vdots \end{array} \end{array} \quad \Leftrightarrow \quad \begin{array}{l} Sys_1(I, I_R) = O \text{ where} \\ \quad \begin{array}{lcl} \vdots & = & \vdots \\ X & = & \alpha(X_R) \\ X_R & = & x_R \mathbin{!} \rho(\alpha(F(X_R, I)), I_R) \\ \vdots & = & \vdots \end{array} \end{array} \tag{130}$$

Proof. It suffices to show that the trace of X is equal in both systems. Proof is by co-recursion on I, I_R. For all $a \in A$:

$$\begin{array}{rcl} head(Tr^*_X[Sys_0](I, a)) & = & \alpha(x) \\ & = & head(Tr^*_X[Sys_1](I, I_R, a)) \end{array} \tag{131}$$

Let $=^*$ denote justification by the co-inductive hypothesis.

$$\begin{array}{lcl} tail(Tr^*_X[Sys_0](I,a)) & = & Tr^*_X[Sys_0](tail(I),\alpha(F(a,head(I)))) \\ & =^* & Tr^*_X[Sys_1](tail(I),tail(I_R),\alpha(F(a,head(I)))) \\ & = & Tr^*_X[Sys_1](tail(I),tail(I_R),\alpha(\rho(\alpha(F(a,head(I))),head(I_R)))) \\ & = & tail(Tr^*_X[Sys_1](I,I_R,a)) \end{array} \tag{132}$$

□

The one-to-many refinement process mirrors the proof steps of the one-to-one refinement theorem, except that the re-embedding transformation (rather than inverse relationship of the coercion operators, ρ and α) justifies step 6 (below). The following steps refine the signal $X = x_0 \; ! \; F(X,I,S,C)$ in a stream system:

1. First introduce the combinational signal X_R, and a sequential signal to hold the next refinement parameter to ρ:

$$\begin{array}{lcl} X & = & x_0 \; ! \; F(X,I,S,C) \\ X_R & = & \rho(X,R) \\ R & = & r_0 \; ! \; \rho(X,R) \end{array} \tag{133}$$

2. Using the term identity $\alpha(\rho(a,r)) = a$, expand the definition of X:

$$\begin{array}{lcl} X & = & \alpha(\rho(x_0,r_0)) \; ! \; \alpha(\rho(F(X,I,S,C),R)) \\ X_R & = & \rho(X,R) \\ R & = & r_0 \; ! \; \rho(X,R) \end{array} \tag{134}$$

3. Rewrite the head and tail terms of X using the homomorphic identities. It is mathematically necessary to perform the rewrites at this stage, since the homomorphic identities are wrapped by the abstraction operator, α:

$$\begin{array}{lcl} X & = & \alpha(\rho(x_0, r_0)) \mathbin{!} \alpha(F_R(\rho(X, R), I, S, C)) \\ X_R & = & \rho(X, R) \\ R & = & r_0 \mathbin{!} \rho(X, R) \end{array} \tag{135}$$

4. Apply combinational identification to X, replacing $\rho(X, R)$ with X_R:

$$\begin{array}{lcl} X & = & \alpha(\rho(x_0, r_0)) \mathbin{!} \alpha(F_R(X_R, I, S, C)) \\ X_R & = & \rho(X, R) \\ R & = & r_0 \mathbin{!} \rho(X, R) \end{array} \tag{136}$$

5. Apply Transformation 6 to convert X_R to a sequential signal:

$$\begin{array}{lcl} X & = & \alpha(\rho(x_0, r_0)) \mathbin{!} \alpha(F_R(X_R, I, S, C)) \\ X_R & = & \rho(\alpha(\rho(x_0, r_0)), r_0) \mathbin{!} \rho(\alpha((F_R(X_R, I, S, C))), R) \\ R & = & r_0 \mathbin{!} \rho(X, R) \end{array} \tag{137}$$

6. Simplify $\mathit{head}(X_R)$ with $\alpha(\rho(x, r)) = x$ and its tail with the re-embedding transformation.

$$\begin{array}{lcl} X & = & \alpha(\rho(x_0, r_0)) \mathbin{!} \alpha(F_R(X_R, I, S, C)) \\ X_R & = & \rho(x_0, r_0) \mathbin{!} F_R(X_R, I, S, C) \\ R & = & r_0 \mathbin{!} \rho(X, R) \end{array} \tag{138}$$

7. Convert X to a combinational signal acting on X_R:

$$\begin{array}{lcl} X & = & \alpha(X_R) \\ X_R & = & \rho(x_0, r_0) \mathbin{!} F_R(X_R, I, S, C) \\ R & = & r_0 \mathbin{!} \rho(X, R) \end{array} \tag{139}$$

8. Eliminate R since it is no longer in use.

$$\begin{array}{lcl} X & = & \alpha(X_R) \\ X_R & = & \rho(x_0, r_0)\ !\ F_R(X_R, I, S, C) \end{array} \tag{140}$$

Example 8.3: This example re-visits the stack calculator refinement with the non-mutating-memory *pop*, and identities defined as in Example 8.2. The one-to-many refinement process produces a system for rewriting. The zero memory m_0 seeds the representation selection signal.

$$\begin{array}{l} StackCalc(instr, a) = res \\ \quad where \\ \quad\quad s^* = *\rho(s, r) \\ \quad\quad s = \alpha(\rho(push(mt,0), r))\ ! \\ \quad\quad\quad\quad \alpha(\rho(sel(instCat(instr), push(s, a), pop(s), \\ \quad\quad\quad\quad\quad\quad push(pop(pop(s)), alu(instr, top(s), top(pop(s))))), r)) \\ \quad\quad res = top(s) \\ \quad\quad r = m_0\ !\ \rho(s, r) \end{array} \tag{141}$$

Rewriting of s proceeds in the manner illustrated by Example 8.1. In particular, the *pop* branch does not zero-out the memory after a decrement:

$$\begin{array}{l} \alpha(sel(\ldots, \rho(pop(s), r), \ldots) = \\ \quad\quad \alpha(sel(\ldots, [\pi_0(\rho(s, r))\ dec(\pi_1(\rho(s, r)))], \ldots) \end{array} \tag{142}$$

All of the leaves of the rewrite have the form $\rho(s, r)$, which get replaced by the refinement signal s^*, following its conversion to sequential form:

$$
\begin{array}{l}
StackCalc(instr, a) = res \\
\quad where \\
\quad\quad s^* \quad = *\rho(s, r) \\
\quad\quad s^* \qquad \equiv \qquad [wr(m_0, 0_{ind}, 0)\ inc(0_{ind})]\ ! \\
\quad\quad\quad\quad\quad\quad sel(instCat(instr), \\
\quad\quad\quad\quad\quad\quad\quad\quad [wr(\pi_0(s^*), \pi_1(s^*), a)\ inc(\pi_1(s^*))] \\
\quad\quad\quad\quad\quad\quad\quad\quad [\pi_0(s^*)\ dec(\pi_1(s^*))] \\
\quad\quad\quad\quad\quad\quad\quad\quad [wr(\pi_0(s^*), dec(dec(\pi_1(s^*))), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad ALUinst(rd(\pi_0(s^*), dec(\pi_1(s^*))), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad rd(\pi_0(s^*), dec(dec(\pi_1(s^*))))))\ dec(\pi_1(s^*))] \\
\quad\quad s \qquad = \qquad \alpha(s^*) \\
\quad\quad res \qquad = \qquad rd(\pi_0(s^*), dec(\pi_1(s^*))) \\
\quad\quad r \qquad = \qquad m_0\ !\ \rho(s, r)
\end{array}
\tag{143}
$$

The final system eliminates the combinational invariant for s^* and the unused signals $\{s, r\}$, then splits the tupled sequential signal s^* into its components, named m and i:

$$
\begin{array}{l}
StackCalc(instr, a) = res \\
\quad where \\
\quad\quad m \quad = \quad wr(m_0, 0_{ind}, 0)\ ! \\
\quad\quad\quad\quad\quad sel(instCat(instr), wr(m, i, a), m, \\
\quad\quad\quad\quad\quad\quad wr(m, dec(dec(i)), ALUinst(rd(m, dec(i)), rd(m, dec(dec(i)))))) \\
\quad\quad i \quad = \quad inc(0_{ind})\ !\ sel(instCat(instr), inc(i), dec(i), dec(i)) \\
\quad\quad res \quad = \quad rd(m, dec(i))
\end{array}
\tag{144}
$$

8.3 Stateful refinement schema

One important class of one-to-many representations are *heap schemas* which represent arbitrary objects as linked trees in a random access store. Tree nodes typically correspond to a data type, such as a LISP pair or atom—e.g., boolean or empty-list.

The *heap* is a linearly ordered random access store which references the first unused cell of the store with a *horizon pointer*. Each time a new object is created the horizon pointer is incremented by an appropriate amount. In real implementations, the store eventually runs out of space and frees up cells in the *garbage collection* process—a re-embedding into a different representation instance that no longer uses cells to represent discarded objects. However, the representations presented here simplify the problem by assuming an unlimited store. This assumption is commonplace in system design and verification [35, 6].

A system defined over abstract data types is often refined into a system which operates on heap implementations of these objects. When the system carries a only one abstract signal, the one-to-many refinement process suffices for a reduction to heap representation. Figure 15 shows a heap representation of stacks. The implementation type is a 3-tuple containing a heap reference, memory of list cells (i.e., tuples of value and address), and a heap horizon pointer. The implementation of *push* writes the input value and the address of input stack the into the cell referenced by the heap horizon; the resulting stack reference is the heap horizon pointer, and finally the heap pointer is incremented. The implementation of *pop* does not alter the heap or horizon, but dereferences the tail of the input stack reference. The implementation of *top* works similarly, dereferencing the value half of the cell referenced by the input stack reference.

Heap implementations of abstract types can simulate multiple signals. While the heap representation of a stack seems wasteful compared to the array-pointer implementation, the same heap may contain multiple independent stacks. Given

Label:	$stack \to heap$
Functions:	$\rho : stack\{integer\} \times [ind\ memory\{ind\ [integer\ ind]\}\ ind]$ $\quad \to [ind\ memory\{ind\ [integer\ ind]\}\ ind]$ $\alpha : [ind\ memory\{ind\ [integer\ ind]\}\ ind] \to stack\{integer\}$
ID Variables:	$s : stack\{integer\},\ d : integer$ $i, r : ind,\ m : memory\{ind\ [integer\ ind]\}$
Identities:	$\mathsf{let}[r^*\ m^*\ i^*] = \rho(s, [r\ m\ i])$ $\alpha(\rho(push(s,d), [r\ m\ i])) = \alpha([i^*\ wr(m^*, i^*, [d\ r^*])\ inc(i^*)])$ $\alpha(\rho(pop(s), [r\ m\ i])) = \alpha([\pi_0(rd(m^* r^*))\ m^*\ i^*])$ $top(s) = \pi_0(rd(m^*, r^*))$ $\rho(mt, [r\ m\ i]) = [0_{ind}\ m*\ i*]$ $\alpha(\rho(s, [r\ m\ i])) = s$

Figure 15: Stack-to-heap refinement declaration.

a schema for how a heap implements one abstract type, one may manipulate the schema to embed many abstract signals into the heap. For example, Figure 15 shows how to implement one stack as a reference to a persistent heap. The heap is *append-only* (the referenced objects in the heap never change), so the same refinement applies element-wise to a tuple of two stacks—or n in general—where heap access is serialized. Refining the two-stack tuple reduces to refining both elements independently. In this case, assume that stack access goes from left to right: refining the second component begins with the heap resulting from the first component's accesses. The following example shows the serialization setup:

$$\begin{array}{lll} & X = mt\ !\ push(X, d_1) & \\ & Y = mt\ !\ push(Y, d_2) & \\ & \Updownarrow & \\ X = \rho(mt, [m_0\ 0]) & ! \ \rho(push(X, d_1), [M\ I]) & \\ Y = \rho(mt, \rho_h(mt, [m_0\ 0])) & ! \ \rho(push(Y, d_2), \rho_h(push(X, d_1), [M\ I])) & \\ [M\ I] = \rho_h(mt, \rho_h(mt, [m_0\ 0])) & ! \ \rho_h(Y, \rho_h(X, [M\ I])) & \end{array} \tag{145}$$

where $[\rho_s\ [\rho_m\ \rho_i]] = [\rho_s\ \rho_h] = \rho$, the hierarchical element structure of the refinement operator. The homomorphic identity declarations for *push* guide the rewriting process. Stateful refinements are formalized below:

Definition 8.2. A *stateful refinement* of an abstract type A has a *reference* component R and a *state* component M. The coercion operators have all the properties of a one-to-many refinement, *and* that the reference component is irrelevant in the refinement selection—i.e.,

$$\forall(a \in A)(m \in M)(r_0, r_1 \in R)\rho(a, [r_0\ m]) = \rho(a, [r_1\ m]) \tag{146}$$

Thus the refinement coercion drops R from its signature, $\rho : A \times M \to R \times M$.

Definition 8.3. Given a stateful coercion operator, $\rho : A \times M \to R \times M$, define a partial order $m_1 \prec' m_2$ if and only if $\pi_1(\rho'(a, m_1)) = m_2$ for some choice of $a \in A$. The transitive closure, $\prec$, of $\prec'$ defines *state-extension*; m_2 extends m_1. Let m be a *valid* state when $m = m_0$ or $m_0 \prec m$. Let $r \in R$ be *valid in* m, written $r \leftarrow m$, when $[r\ m] = \pi_1(\rho'(a, m'))$ for some valid state m'. Then ρ' *preserves prior references* when $r \leftarrow m_1$ and $m_1 \prec m_2$ imply $\alpha(r, m_1) = \alpha(r, m_2)$.

Heap representations are stateful refinements: The store and horizon pointer completely determine the implementation of abstract types. Heaps preserve prior references since they are append-only—they are not garbage collected in the model presented here.

Definition 8.4. A function in a stateful representation $G : R^n \times M \times T \to R^n \times M$ is *referentially independent* when

$$\alpha(G(r_1, \ldots, r_n, m, t)) = \alpha(G(r'_1, \ldots, r'_n, m', t)) \tag{147}$$

where $\alpha(r_i, m) = \alpha(r'_i, m')$ for $1 \leq i \leq n$. Referential independence in a stateful implementation function asserts that abstractions of its output are independent of the input representations (since there are many).

The general case of multiple signal refinement is difficult to argue because of the exploding number of nested and repeated relevant terms. However, the two signal case uses the same core justifications and is much more illuminating. To further simplify matters, we assume that there is only one abstract function, and it takes inputs from the abstract type. There is little loss of generality since this function may be parameterized by a token which calls an arbitrary one or two input function in the signature.

Proposition 8.1. *(Two signal Stateful Refinement) Let* $\rho : A \times M \to R \times M$ *and* $\alpha : R \times M \to A \times M$ *be stateful coercion operators that preserve prior references. Let* $[\rho_R, \rho_M] = \rho$*. Assume that the refinement's identities can affect the following rewrite for* $1 \leq i \leq n$*:*

$$\alpha(\rho(F(A_1, A_2, I, S, C), M)) = \alpha(G(\rho_R(A_1, M), \rho_R(A_2, \rho_R(M)), M, I, S, C)) \tag{148}$$

where G *is referentially independent. The right-hand-side serializes access to the store in a left-to-right manner. The function operates on the two references, resulting store, and non-subject-type values. The following are equivalent systems by pointwise*

alignment:

$$
\begin{array}{l}
\begin{array}{l}
Sys_0(I) = O \ \textit{where} \\
\quad \begin{array}{lcl}
\vdots & = & \vdots \\
A_1 & = & a_1 \ ! \ F(A_1, A_2, I_1, S_1, C_1) \\
A_2 & = & a_2 \ ! \ F(A_1, A_2, I_2, S_2, C_2) \\
\vdots & = & \vdots
\end{array}
\end{array} \\
\qquad\qquad \Updownarrow \\
\begin{array}{l}
Sys_1(I) = O \ \textit{where} \\
\quad \begin{array}{lcl}
\vdots & = & \vdots \\
A_1 & = & \alpha(R_1, M) \\
A_2 & = & \alpha(R_2, M) \\
R_1 & = & \rho_R(a_1, m_0) \ ! \ \pi_R(G(R_1, R_2, M, I_1, S_1, C_1)) \\
R_2 & = & \rho_R(a_2, \rho_m(a_1, m_0)) \ ! \ \pi_R(G(R_1, R_2, M^*, I_2, S_2, C_2)) \\
M & = & \rho_M(a_2, \rho_M(a_1, m_0)) \ ! \ \pi_M(G(R_1, R_2, M^*, I_2, S_2, C_2)) \\
M^* & = & \pi_M(G(R_1, R_2, M_2, I_1, S_1, C_1)) \\
\vdots & = & \vdots
\end{array}
\end{array}
\end{array}
\tag{149}
$$

Proof. Introduce signals $A_1^*, A_2^*, M_0, [R_1 \ M_1], [R_2 \ M_2], M_3, M_4$. Store operations are serialized, so that $M_0 \prec M_1 \prec M_2 \prec M_3 \prec M_4$. Move sequential updates for A_1, A_s into A_1^*, A_2^*:

$$
\begin{array}{lcl}
A_1 & = & a_1 \ ! \ A_1^* \\
A_2 & = & a_2 \ ! \ A_2^* \\
A_1^* & = & F(A_1, A_2, I_1, S_1, C_1) \\
A_2^* & = & F(A_1, A_2, I_2, S_2, C_2) \\
M_0 & = & m_0 \ ! \ M_4 \\
[R_1 \ M_1] & = & \rho(A_1, M_0) \\
[R_2 \ M_2] & = & \rho(A_2, M_1) \\
M_3 & = & \pi_M(G(R_1, R_2, M_2, I_1, S_1, C_1)) \\
M_4 & = & \pi_M(G(R_1, R_2, M_3, I_2, S_2, C_2))
\end{array}
\tag{150}
$$

Expand A_1 by the abstraction identity, $alpha(\rho(a, m)) = a$:

$$\begin{array}{lcl} A_1 & = & a_1 \mathbin{!} A_1^* \\ A_2 & = & a_2 \mathbin{!} A_2^* \\ A_1^* & = & \alpha(\rho(F(A_1, A_2, I_1, S_1, C_1), M_2)) \\ A_2^* & = & F(A_1, A_2, I_2, S_2, C_2) \\ M_0 & = & m_0 \mathbin{!} M_4 \\ [R_1\ M_1] & = & \rho(A_1, M_0) \\ [R_2\ M_2] & = & \rho(A_2, M_1) \\ M_3 & = & \pi_M(G(R_1, R_2, M_2, I_1, S_1, C_1)) \\ M_4 & = & \pi_M(G(R_1, R_2, M_3, I_2, S_2, C_2)) \end{array} \tag{151}$$

Rewrite A_1 according to homomorphic assumption (148):

$$\begin{array}{lcl} A_1 & = & a_1 \mathbin{!} A_1^* \\ A_2 & = & a_2 \mathbin{!} A_2^* \\ A_1^* & = & \alpha(G(\rho_R(A_1, M_2), \rho_R(A_2, \rho_R(A_1, M_2)), \rho_M(A_2, \rho_R(A_1, M_2)), I_1, S_1, C_1)) \\ A_2^* & = & F(A_1, A_2, I_2, S_2, C_2) \\ M_0 & = & m_0 \mathbin{!} M_4 \\ [R_1\ M_1] & = & \rho(A_1, M_0) \\ [R_2\ M_2] & = & \rho(A_2, M_1) \\ M_3 & = & \pi_M(G(R_1, R_2, M_2, I_1, S_1, C_1)) \\ M_4 & = & \pi_M(G(R_1, R_2, M_3, I_2, S_2, C_2)) \end{array} \tag{152}$$

Since $M_0 \prec M_2$ (hence $\alpha(\rho(A_1, M_2)) = \alpha(\rho(A_1, M_0))$) and G is referentially independent, replace $\rho_R(A_1, M_2)$ in A_1^* with $\rho_R(A_1, M_0)$ and then with R_1 by combinational identification. Similarly $M_2 \prec \rho_R(A_1, M_2)$ justifies replacing G's second argument

with R_2.

$$\begin{array}{lcl} A_1 & = & a_1 \mathbin{!} A_1^* \\ A_2 & = & a_2 \mathbin{!} A_2^* \\ A_1^* & = & \alpha(G(R_1, R_2, M_2, I_1, S_1, C_1)) \\ A_2^* & = & F(A_1, A_2, I_2, S_2, C_2) \\ M_0 & = & m_0 \mathbin{!} M_4 \\ [R_1\ M_1] & = & \rho(A_1, M_0) \\ [R_2\ M_2] & = & \rho(A_2, M_1) \\ M_3 & = & \pi_M(G(R_1, R_2, M_2, I_1, S_1, C_1)) \\ M_4 & = & \pi_M(G(R_1, R_2, M_3, I_2, S_2, C_2)) \end{array} \tag{153}$$

The same logic transforms A_2^*:

$$\begin{array}{lcl} A_1 & = & a_1 \mathbin{!} A_1^* \\ A_2 & = & a_2 \mathbin{!} A_2^* \\ A_1^* & = & \alpha(G(R_1, R_2, M_2, I_1, S_1, C_1)) \\ A_2^* & = & \alpha(G(R_1, R_2, M_3, I_2, S_2, C_2)) \\ M_0 & = & m_0 \mathbin{!} M_4 \\ [R_1\ M_1] & = & \rho(A_1, M_0) \\ [R_2\ M_2] & = & \rho(A_2, M_1) \\ M_3 & = & \pi_M(G(R_1, R_2, M_2, I_1, S_1, C_1)) \\ M_4 & = & \pi_M(G(R_1, R_2, M_3, I_2, S_2, C_2)) \end{array} \tag{154}$$

Prepare $[R_2\ M_2]$ for conversion to sequential signals by expanding combinational reference to M_1:

$$\begin{array}{lcl} A_1 & = & a_1 \mathbin{!} \alpha(G(R_1, R_2, M_2, I_1, S_1, C_1)) \\ A_2 & = & a_2 \mathbin{!} \alpha(G(R_1, R_2, M_3, I_2, S_2, C_2)) \\ M_0 & = & m_0 \mathbin{!} M_4 \\ [R_1\ M_1] & = & \rho(A_1, M_0) \\ [R_2\ M_2] & = & \rho(A_2, \rho_M(A_1, M_0)) \\ M_3 & = & \pi_M(G(R_1, R_2, M_2, I_1, S_1, C_1)) \\ M_4 & = & \pi_M(G(R_1, R_2, M_3, I_2, S_2, C_2)) \end{array} \tag{155}$$

Convert $[R_1\ M_1], [R_2\ M_2]$ to sequential signals:

$$\begin{array}{lcl}
A_1 & = & a_1\ !\ \alpha(G(R_1, R_2, M_2, I_1, S_1, C_1)) \\
A_2 & = & a_2\ !\ \alpha(G(R_1, R_2, M_3, I_2, S_2, C_2)) \\
M_0 & = & m_0\ !\ M_4 \\
[R_1\ M_1] & = & \rho(a_1, m_0)\ ! \\
 & & \quad \rho(\alpha(G(R_1, R_2, M_2, I_1, S_1, C_1)), M_4) \\
[R_2\ M_2] & = & \rho(a_2, \rho_M(a_1, m_0))\ ! \\
 & & \quad \rho(\alpha(G(R_1, R_2, M_3, I_2, S_2, C_2)), \rho_M(A_1, M_4)) \\
M_3 & = & \pi_M(G(R_1, R_2, M_2, I_1, S_1, C_1)) \\
M_4 & = & \pi_M(G(R_1, R_2, M_3, I_2, S_2, C_2))
\end{array} \tag{156}$$

Apply re-embedding transformation.

$$\begin{array}{lcl}
A_1 & = & a_1\ !\ \alpha(G(R_1, R_2, M_2, I_1, S_1, C_1)) \\
A_2 & = & a_2\ !\ \alpha(G(R_1, R_2, M_3, I_2, S_2, C_2)) \\
M_0 & = & m_0\ !\ M_4 \\
[R_1\ M_1] & = & \rho(a_1, m_0)\ !\ G(R_1, R_2, M_2, I_1, S_1, C_1) \\
[R_2\ M_2] & = & \rho(a_2, \rho_M(a_1, m_0))\ !\ G(R_1, R_2, M_3, I_2, S_2, C_2) \\
M_3 & = & \pi_M(G(R_1, R_2, M_2, I_1, S_1, C_1)) \\
M_4 & = & \pi_M(G(R_1, R_2, M_3, I_2, S_2, C_2))
\end{array} \tag{157}$$

Expand head terms in A_1, A_2:

$$\begin{array}{lcl}
A_1 & = & \alpha(\rho(a_1, m_0))\ !\ \alpha(G(R_1, R_2, M_2, I_1, S_1, C_1)) \\
A_2 & = & \alpha(\rho(a_2, \rho(a_1, m_0)))\ !\ \alpha(G(R_1, R_2, M_3, I_2, S_2, C_2)) \\
M_0 & = & m_0\ !\ M_4 \\
[R_1\ M_1] & = & \rho(a_1, m_0)\ !\ G(R_1, R_2, M_2, I_1, S_1, C_1) \\
[R_2\ M_2] & = & \rho(a_2, \rho_M(a_1, m_0))\ !\ G(R_1, R_2, M_3, I_2, S_2, C_2) \\
M_3 & = & \pi_M(G(R_1, R_2, M_2, I_1, S_1, C_1)) \\
M_4 & = & \pi_M(G(R_1, R_2, M_3, I_2, S_2, C_2))
\end{array} \tag{158}$$

Convert A_1, A_2 to combinational signals:

$$\begin{array}{lcl} A_1 & = & \alpha([R_1\ M_1]) \\ A_2 & = & \alpha([R_2\ M_2]) \\ M_0 & = & m_0\ !\ M_4 \\ [R_1\ M_1] & = & \rho(a_1, m_0)\ !\ G(R_1, R_2, M_2, I_1, S_1, C_1) \\ [R_2\ M_2] & = & \rho(a_2, \rho_M(a_1, m_0))\ !\ G(R_1, R_2, M_3, I_2, S_2, C_2) \\ M_3 & = & \pi_M(G(R_1, R_2, M_2, I_1, S_1, C_1)) \\ M_4 & = & \pi_M(G(R_1, R_2, M_3, I_2, S_2, C_2)) \end{array} \tag{159}$$

Since $M_1 \prec M_2$, replace M_1 with M_2 in A_1. Ungroup $[R_1\ M_1], [R_2\ M_2]$, then eliminate M_0, M_1, M_4:

$$\begin{array}{lcl} A_1 & = & \alpha([R_1\ M_1]) \\ A_2 & = & \alpha([R_2\ M_2]) \\ R_1 & = & \rho_R(a_1, m_0)\ !\ \pi_R(G(R_1, R_2, M_2, I_1, S_1, C_1)) \\ R_2 & = & \rho_R(a_2, \rho_M(a_1, m_0))\ !\ \pi_R(G(R_1, R_2, M_3, I_2, S_2, C_2)) \\ M_2 & = & \rho_M(a_2, \rho_M(a_1, m_0))\ !\ \pi_M(G(R_1, R_2, M_3, I_2, S_2, C_2)) \\ M_3 & = & \pi_M(G(R_1, R_2, M_2, I_1, S_1, C_1)) \end{array} \tag{160}$$

Renaming signal M_2 as M and M_3 as M^* completes the proof.

□

As the resulting expression shows, even the two-signal stateful refinement can accumulate deeply nested store expressions. The problem is mitigated by using combinational signals as temporary bindings.

8.4 Starfish's refinement provisions

Starfish supports refinement with declarations for coercion operators and related identities, application of the signal refinement transformation to behavior tables, and

automatic rewriting of terms with respect to the refinement homomorphism. Refinement declarations enumerate the abstract and refinement types, homomorphic identities for each function that produces or consumes the abstract type, and the mapping of abstract constant terms. Starfish's refinement transformation applies the signal refinement theorem, rewriting the coercion terms according to the homomorphic identities. In the case of stateful refinement schema, the declarations additionally define state types, and initial state values. Starfish uses the following syntax for refinement declarations:

```
(declare-refinement <src-sym> => <tar-sym>
   (<param-sym> ...)
   <src-type-str>
   <tar-type-str>
   ((<state-sym> <type-str> <state-inital-value-str>) ...)
   ((<formal-sym> <type-str>)...)
   ((<binding-sym> <exp-str>) ...)
   ((<hom-id-src-str> <hom-id-tar-str> <hom-id-state-update-str> ...) ...)
   ((<hom-id-label-sym> <hom-id-src-str> <hom-id-tar-str>) ...)
   ((<id-label-sym> <id-src-str> <id-tar-str>) ...))
```

The `<src-sym>` and `<tar-sym>` symbols create a shorthand name for the refinement (`<src-sym>=><tar-sym>`), and names for the coercion operators—`<src-sym>=><tar--sym>` for refinement and `<src-sym><=<tar-sym>` for abstraction. If the refinement is for an uninstantiated parameterized type, the parameter symbols are listed by `(<param-sym> ...)`. The fields `<src-type-str>` and `<tar-type-str>` specify the abstract and refinement types. When defining a multiple reference to state schema `((<state-sym> <type-str> <state-inital-value-str>) ...)` defines the state labels, types, and intial values (e.g., heap states consist of a memory and a pointer with initial values of m_0 and 0_{ind}). The remaining sections specify homomorphic identities using the typed formal variables in `((<formal-sym> <type-str>)...)`

The homomorphic declarative form focuses on term expressions which define the abstract and implementation functions. Bindings in `((<binding-sym> <exp-str>) ...)` assist the functional description for the identity list, `(<hom-id-src-str> <hom-id-tar-str>)`. For example, a homomorphic identity $\alpha(\rho(f(a))) = \alpha(g(\rho(a)))$ might be expressed as `((f a) (g r))` in context of the binding `(r (src=>tar a))`.

Stateful rewriting rules explicitly decompose into references and stores. Since the store is often a tuple of types, the declarative form permits component expression of state rather than forcing obfuscation with a single tupled state. Heaps, for instance, track state with a linearly ordered random access store and a horizon pointer. The homomorphic identity which serialized effects on the store

$$\alpha(\rho(f(a), [mem\ ptr])) = \alpha(g(\rho_R(a), [h(\rho_{mem}(a, [mem\ ptr]))\ inc(\rho_{mem}(a, [mem\ ptr]))])) \tag{161}$$

is expressed as

```
...
 ; Bindings
 ((r (src=>tar a mem ptr))
  (mem* (src=>tar.mem a mem ptr))
  (ptr* (src=>tar.ptr a mem ptr)))
 ; Homomorphic Identities
 (...
  [(f a)
   (g r) (h mem*) (inc ptr*)] ...)
...
```

The binding list `((<binding-sym> <exp-str>) ...)` is similar to Scheme's `let*`, where each bound symbol is in the scope of the following expression. In this declarative form for homomorphic identities, the state arguments and coercion operator is implied on the left-hand side; e.g., `f(a)` specifies `(src=>tar f(a) mem ptr)`. All of

the rewrites occur inside of an abstraction operator, where they are valid.

Starfish automates rewriting in data refinement. It attempts to match each coercion subterm with one of the identities from the list

```
(<hom-id-src-str> <hom-id-tar-str> <hom-id-state-update-str>  ...)
```

The declaration should have a unique left-hand-side for each function which produces the abstract type. The identities should be "homomorphic" in the sense that the right-hand-side is an expression that only applies the refinement operator to the abstract arguments of left-hand-side's function. The leaves of the rewrite apply the refinement operator to the sequential signal variable of the abstract type. As shown in the refinement proofs, replacing these with the refinement type's signal variable produces a bisimilar system.

The next clause of the declaration,

```
((<hom-id-label-sym> <hom-id-src-str> <hom-id-tar-str>) ...)
```

expresses identities on functions that consume, but do not produce abstract types. They are also in scope of the binding list. For example, the declaration

```
...
 ; Bindings
 ((r (src=>tar a mem ptr))
  (mem* (src=>tar.mem a mem ptr))
  (ptr* (src=>tar.ptr a mem ptr)))
 ; Homomorphic Identities
 (...
  [(f a t)
   (g r mem* ptr* t)] ...)
...
```

expresses the identity $f(\rho(a, [mem\ ptr]), t) = g(\rho(a, [mem\ ptr]), t)$ Starfish automates rewriting for these terms as well.

The last list of the declaration

```
((<id-label-sym> <id-src-str> <id-tar-str>) ...)
```

holds auxiliary labeled identities in equational form. There are no implied operators on left- or right-hand sides; `<id-src-str>` is the left-hand-side, `<id-tar-str>` is the right-hand-side. This last set of supplemental identities are *not* used in automated term rewriting. The designer applies them manually, specifying them with `<id-label-sym>`.

Example 8.4: The following syntax expresses the optimized array-with-pointer stack refinement to Starfish.

```
(declare-refinement stack => array
  () ; no parameters
  "stack{integer}"
  "[memory{ind integer} ind]"
  () ; no state
  ;; Formals
  ([s "stack{integer}"]
   [d "integer"])
  ;; Bindings
  ([m "(1st (stack=>array s))"]
   [i "(2nd (stack=>array s))"])
  ;; Homomorphic identities for functions that produce stacks
  (["(push s d)" "[(wr m i d) (inc i)]"]
   ["(pop s)" "[m (dcr i)]"]
   ["empty-stack:stack{integer}" "[m0:memory{ind integer} 0_ind]"])
  ;; Homomorphic identities for functions that consume
  ;; but do not produce stacks
  (['stack=>array:top "(top s)" "(rd m (dcr i))"])
  ;; Supplemental identities
  ())
```

Although stacks are a parameterized type, this refinement declaration is for stacks of integers; consequently, the list of type parameters is empty. The declaration specifies `stack{integer}` as the abstract type and `[memory{ind integer} ind]` as the refinement type. This is not a reference-to-state schema, so the list of state labels, types

and initial values is empty. Two formal variables express the homomorphic identities: stack `s` and stack datum `d`. The symbols `m` and `i` expand to the memory component and pointer component respectively of `s`'s refinement. The clause after the bindings shows the homomorphic identities for the stack producing functions `push` and `pop`. The next clause specifies rewrites for `top` which returns an integer instead of a stack. No supplemental identities are listed here.

Example 8.5: The following syntax expresses the multi-reference heap refinement schema for integer stacks to Starfish:

```
(declare-refinement stack => heap
  () ; no parameters
  "stack{integer}"
  "ind" ; reference type
  ; state symbol, type, and initial values
  ([mem "memory{ind [integer ind]}" "#:memory{ind [integer ind]}"]
   [ptr "ind" "0_ind"])
  ; Formals
  ([s "stack{integer}"]
   [d "integer"])
  ;; Bindings
  ([s* "(stack=>heap s mem ptr)"]
   [mem* "(stack=>heap.mem s mem ptr)"]
   [ptr* "(stack=>heap.ptr s mem ptr)"])
  ;; Homomorphic identities for functions that produce stacks
  (["(push s d)"
    "ptr*" "(wr mem* ptr* [d s*])" "(inc ptr*)"]
   ["(pop s)"
    "(2nd (rd mem* s*))" "mem*" "ptr*"]
   ["empty-stack:stack{integer}" "0_ind" "mem*" "ptr*"])
  ;; Homomorphic identities for functions that consume
  ;; but do not produce stacks
  (['stack=>heap:top "(top s)""(1st (rd mem* s*))"])
  ;; Supplemental Identities
  ())
```

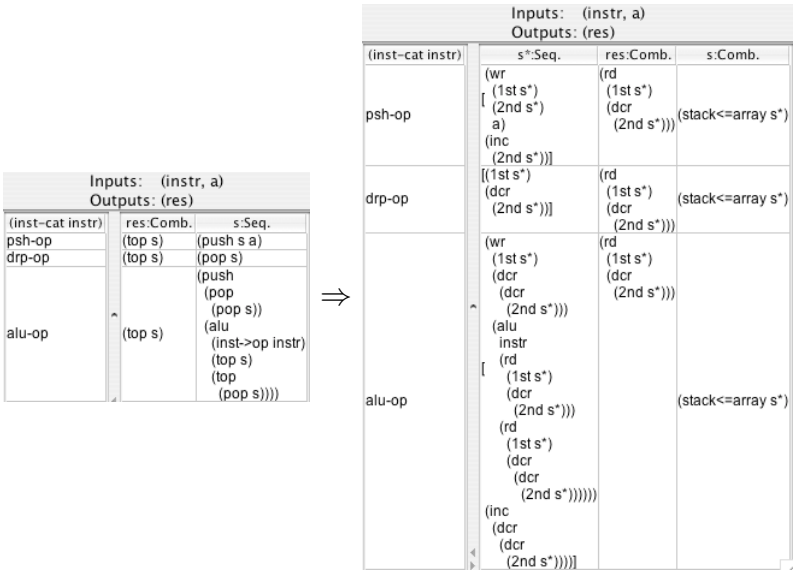
In the reference-state schema the `<tar-type-str>` defines the reference type, `ind`.

The next clause expresses state labels (for reference in identity declarations), state types, and each state signal's initial value. When developing the state schema in Section 8.3, the state component is a single signal. In pratice the state signal is often a tuple. This declarative form facilitates state expression by listing tuple components as separate signals. For heaps, the state consists of a memory *and* a heap pointer. These components are declared separately with labels `mem` and `ptr`, rather than as a single tupled state. The functions `stack=>heap`, `stack=>heap.mem`, and `stack=>heap.ptr` with input signatures $ind \times memory\{ind\ [integer\ ind]\} \times ind$ produce the three components of the refinement—i.e., the reference, the memory, and the heap pointer, respectively; the refinement automatically generates these operators using the `<src-sym>`, `<tar-sym>`, and `<state-sym>` fields.

The binding convention indicates the components of `s`'s implementation with `*`-suffixed labels. `m*` and `ptr*` represent the components of the post-`s` refinement state. Intuitively, this is the *current* state in the subsequent identities. Identity declaration for the non-stack-producer, `top`, eliminates the expressions for state update. The declarative form disallows state effects when no new stack is produced.

Starfish's refinement transformation combines the system level signal introductions from the refinement theorem, automatic rewriting of terms to elimiate coercion operators, and specialization of non-equals refinement signals to equals stream equations. The rewriting algorithm recursively applys the homomorphic identities from the refinement declaration, simplifies projections, and replaces $\rho(a)$ with r where a is the abstract target signal and r is the refinement signal.

Example 8.6: The following behavior tables show the action of the `stack=>array` refinement from Example 8.4 on the stack calculator specification. This is the behavior table version of the stream system refinement in Example 8.3. The refinement command specifies the abstract signal `s`, the name of the refinement signal `s*`, that the automatic rewriting should extend to the subterms of `res`, and refinement declaration `stack=>array`:



Inputs: (instr, a)
Outputs: (res)

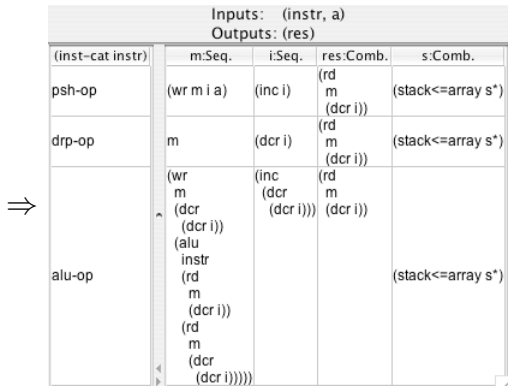
(inst-cat instr)	res:Comb.	s:Seq.
psh-op	(top s)	(push s a)
drp-op	(top s)	(pop s)
alu-op	(top s)	(push (pop (pop s)) (alu (inst->op instr) (top s) (top (pop s))))

⇒

Inputs: (instr, a)
Outputs: (res)

(inst-cat instr)	s*:Seq.	res:Comb.	s:Comb.
psh-op	[(wr (1st s*) (2nd s*) a) (inc (2nd s*))]	(rd (1st s*) (dcr (2nd s*)))	(stack<=array s*)
drp-op	[(1st s*) (dcr (2nd s*))]	(rd (1st s*) (dcr (2nd s*)))	(stack<=array s*)
alu-op	[(wr (1st s*) (dcr (dcr (2nd s*))) (alu instr (rd (1st s*) (dcr (2nd s*))) (rd (1st s*) (dcr (dcr (2nd s*)))))) (inc (dcr (dcr (2nd s*))))]	(rd (1st s*) (dcr (2nd s*)))	(stack<=array s*)

Since the implementation type is a tuple of memory and index, the resulting projectors obfuscate the terms. Splitting the implementation signal into components named `m` and `i` simplifies the update terms:



⇒

Inputs: (instr, a)
Outputs: (res)

(inst-cat instr)	m:Seq.	i:Seq.	res:Comb.	s:Comb.
psh-op	(wr m i a)	(inc i)	(rd m (dcr i))	(stack<=array s*)
drp-op	m	(dcr i)	(rd m (dcr i))	(stack<=array s*)
alu-op	(wr m (dcr (dcr i)) (alu instr (rd m (dcr i)) (rd m (dcr (dcr i)))))	(inc (dcr (dcr i)))	(rd m (dcr i))	(stack<=array s*)

Example 8.7: The following behavior tables show the action of the `stack=>heap` refinement from Example 8.5 on the stack calculator specification.

Prior to stack refinement, this example adds a combinational signal `tl` to hold the term `(pop s)`.

Inputs: (instr, a)
Outputs: (res)

(inst-cat instr)	res:Comb.	s:Seq.	tl:Comb.
psh-op	(top s)	(push s a)	(pop s)
drp-op	(top s)	(pop s)	(pop s)
alu-op	(top s)	(push (pop (pop s)) (alu instr (top s) (top (pop s))))	(pop s)

The update actions for `s` simplify using the combinational identity `tl=(pop s)`.

Inputs: (instr, a)
Outputs: (res)

(inst-cat instr)	res:Comb.	s:Seq.	tl:Comb.
psh-op	(top s)	(push s a)	(pop s)
drp-op	(top s)	tl	(pop s)
alu-op	(top s)	(push (pop tl) (alu instr (top s) (top tl)))	(pop s)

The next step transforms `tl` from a combinational signal to a sequential signal.

Inputs: (instr, a)
Outputs: (res)

(inst-cat instr)	res:Comb.	s:Seq.	tl:Seq.
psh-op	(top s)	(push s a)	(pop (push s a))
drp-op	(top s)	tl	(pop tl)
alu-op	(top s)	(push (pop tl) (alu instr (top s) (top tl)))	(pop (push (pop tl) (alu instr (top s) (top tl))))

The stack identity $pop(push(s, d)) = s$ simplifies the actions for `tl` in the `psh-op` and `alu-op` cases.

Inputs: (instr, a)
Outputs: (res)

(inst-cat instr)	res:Comb.	s:Seq.	tl:Seq.
psh-op	(top s)	(push s a)	(pop (push s a))
drp-op	(top s)	tl	(pop tl)
alu-op	(top s)	(push (pop tl) (alu instr (top s) (top tl)))	(pop (push (pop tl) (alu instr (top s) (top tl))))

The data refinement represents `s` and `tl` with two reference signals, `s*` and `tl*`, and two shared state signals, `m` and `i`.

Inputs: (instr, a)
Outputs: (res)

(inst-cat instr)	tl*:Seq.	s*:Seq.	mem:Seq.	ptr:Seq.	res:Comb.
psh-op	tl*	ptr	(wr mem ptr [a tl*])	(inc ptr)	(1st (rd mem tl*))
drp-op	(2nd (rd mem s*))	s*	mem	ptr	(1st (rd mem tl*))
alu-op	(2nd (rd mem s*))	ptr	(wr mem ptr (alu instr [(1st (rd mem tl*)) (1st (rd mem s*))) (2nd (rd mem s*))]))	(inc ptr)	(1st (rd mem tl*))

The multi reference refinement command specifies the table in the tree hierarchy (top level in this case), the abstract signals `s` and `tl`, names for the reference signals `s*`

and `tl*`, names for each of the shared state signals `mem` and `ptr`, that the automatic rewriting should extend to the subterms of `res`, and the refinement declaration `stack=>heap`:

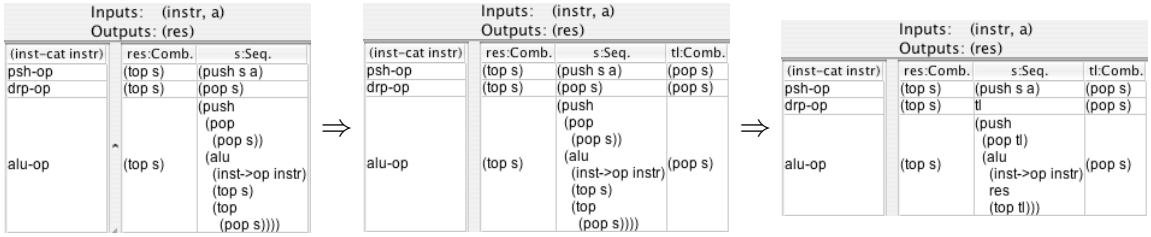
```
(apply-data-refinement (make-sys-path)
  '((tl . tl*) (s . s*)) '(mem ptr) '(res) 'stack=>heap)
```

The order of abstract signal specification also prescribes the order of state access. In this example, `tl` updates the state before `s`.

8.5 Putting it all together

This derivation combines factorization, serialization, data refinement, and retiming to produce a high level architecture for the abstract stack calculator specification. The target architecture represents stacks with a heap. It has several registers: stack top value, reference to stack tail, reference to stack, reference to heap horizon, and a ALU result buffer. The derivation factors out the memory and ALU. The controller is a collection of register transfers between inputs and its internal registers. The controller applies only trivial functions—i.e., those implementable with bit-vector masks in the target medium. Figure 16 illustrates the specification and decomposition with architectural diagrams.

Step 1



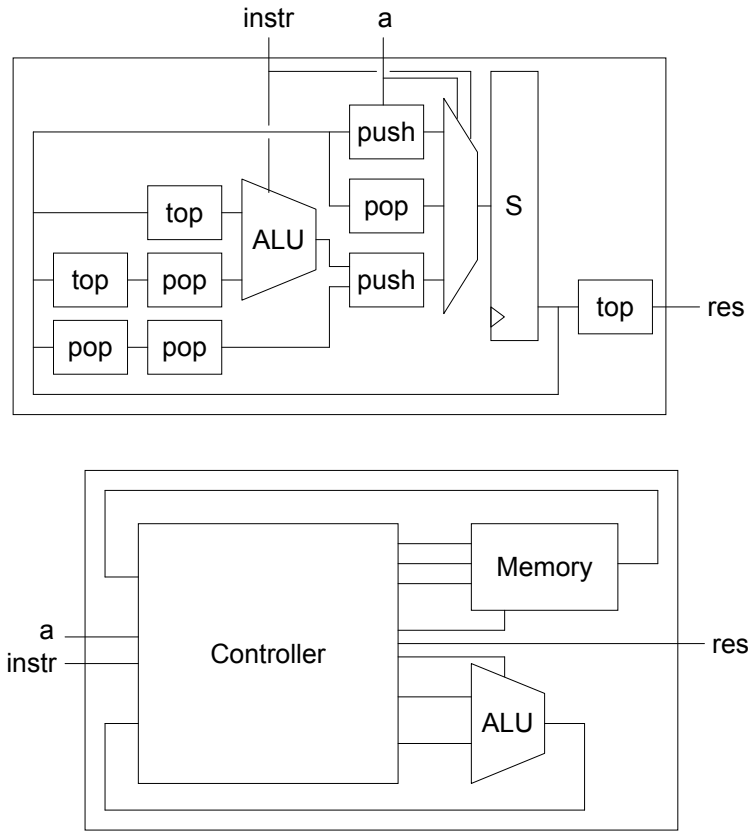
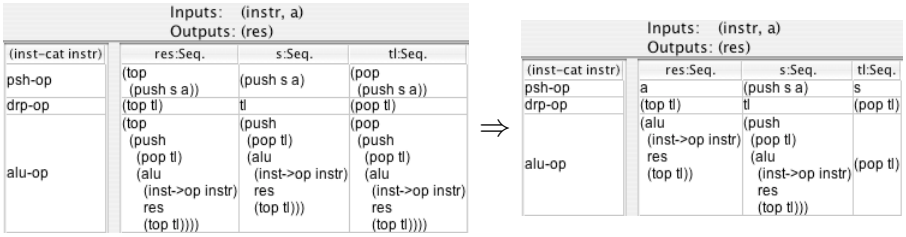


Figure 16: The top diagram is an architectural view of the specification behavior table. The bottom diagram is the resulting decomposition, where the controller is a register-transfer machine.

The first step is to introduce a combinational signal `tl` that constantly holds the value `(pop s)`. The derivation simplifys the action terms of `s` by replacing `(pop s)` and `(top s)` with `tl` and `res`, respectively. It is critical to make these substitutions now, because translation to sequential signals eliminates Starfish's access to the invariants `tl = (pop s)` and `res = (top s)`.

Step 2



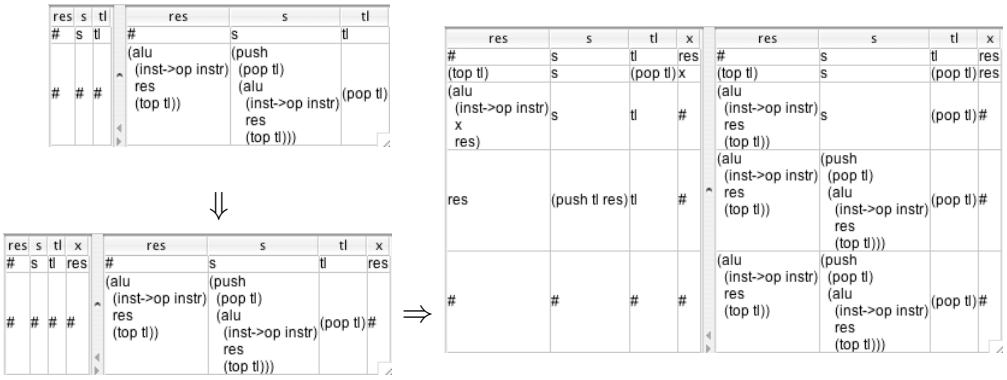
Inputs: (instr, a) Outputs: (res)			
(inst-cat instr)	res:Seq.	s:Seq.	tl:Seq.
psh-op	(top (push s a))	(push s a)	(pop (push s a))
drp-op	(top tl)	tl	(pop tl)
alu-op	(top (push (pop tl) (alu (inst->op instr) res (top tl))))	(push (pop tl) (alu (inst->op instr) res (top tl)))	(pop (push (pop tl) (alu (inst->op instr) res (top tl))))

⇒

Inputs: (instr, a) Outputs: (res)			
(inst-cat instr)	res:Seq.	s:Seq.	tl:Seq.
psh-op	a	(push s a)	s
drp-op	(top tl)	tl	(pop tl)
alu-op	(alu (inst->op instr) res (top tl))	(push (pop tl) (alu (inst->op instr) res (top tl)))	(pop tl)

The signals `res` and `tl` are candidates for register conversion because their actions do not use either of the input signals. Unrolling the combinational signals opens many opportunities for term simplification; immediate `top` and `pop` accesses after a `push` recover the `push` value and old stack respectively. These simplifications are applied to produce the table on the above, right. Recall that the solution streams for `res` and `tl` are unaltered. So the invariants `tl = (pop s)` and `res = (top s)` still hold. Since the combinational representation of the signals is replaced, the behavior table algebra does not retain these invariants.

Step 3



res	s	tl
#	s	tl
#	#	#

res	s	tl
#	s	tl
(alu (inst->op instr) res (top tl))	(push (pop tl) (alu (inst->op instr) res (top tl)))	(pop tl)

⇓

res	s	tl	x
#	s	tl	res
#	#	#	#

res	s	tl	x
#	s	tl	res
(alu (inst->op instr) res (top tl))	(push (pop tl) (alu (inst->op instr) res (top tl)))	(pop tl)	#

⇒

res	s	tl	x
#	s	tl	res
(top tl)	s	(pop tl)	x
(alu (inst->op instr) x res)	s	tl	#
res	(push tl res)	tl	#
#	#	#	#

res	s	tl	x
#	s	tl	res
(top tl)	s	(pop tl)	res
(alu (inst->op instr) res (top tl))	s	(pop tl)	#
(alu (inst->op instr) res (top tl))	(push (pop tl) (alu (inst->op instr) res (top tl)))	(pop tl)	#
(alu (inst->op instr) res (top tl))	(push (pop tl) (alu (inst->op instr) res (top tl)))	(pop tl)	#

The above sequence develops the serialization table for the action guarded by `alu-op`. The subject signals are `res`, `s`, and `tl`. The above left is the initial table. Recall that the LHS contains the proposed serial sub-actions, the RHS holds the serial symbolic

evaluation of the LHS, and that the last row shows the evaluation targets in the RHS (and nothing in the LHS). The next table introduces a new registered signal, `x`, to temporarily hold the value of `res`. The final table shows the rest of the serialization proposal. Since the result of symbolic evaluation matches the evaluation targets, this is a valid serialization.

Step 4

The left table below inserts the schedule in the behavior table.

Inputs: (instr, a)
Outputs: (res)

ser	(inst-cat instr)	res:Seq.	s:Seq.	tl:Seq.	x:Seq.
#	psh-op	a	(push s a)	s	#
#	drp-op	(top tl)	tl	(pop tl)	#
0	alu-op	#	s	tl	res
1	alu-op	(top tl)	s	(pop tl)	x
2	alu-op	(alu (inst->op instr) x res)	s	tl	#
3	alu-op	res	(push tl res)	tl	#

⇒

Inputs: (instr, a)
Outputs: (res)

ser	(inst-cat instr)	s*:Seq.	tl*:Seq.	mem:Seq.	ptr:Seq.	res:Seq.	x:Seq.
#	psh-op	ptr	s*	(wr mem ptr [a s*])	(inc ptr)	a	#
#	drp-op	tl*	(2nd (rd mem tl*))	mem	ptr	(1st (rd mem tl*))	#
0	alu-op	s*	tl*	mem	ptr	#	res
1	alu-op	s*	(2nd (rd mem tl*))	mem	ptr	(1st (rd mem tl*))	x
2	alu-op	s*	tl*	mem	ptr	(alu (inst->op instr) x res)	#
3	alu-op	ptr	tl*	(wr mem ptr [res tl*])	(inc ptr)	res	#

The right table (above) shows the result of data refinement from abstract stacks to heaps. The two stack signals `s` and `tl` are refined to reference signals `s*` and `tl*`, using the refinement identities defined by Figure 15. The signal `mem` is the infinite heap and `ptr` points to the next free cell. This is an example of the many-to-one stateful refinement schema from Section 8.3.

Step 5

The `ptr` signal has type `ind`. This is an attractive signal for `integer` refinement becuase the `inc` operation could make use of `alu` evaluation. One of the derivaiton goals is to factor out a separate ALU; using this apparatus to aid stack management is a desirable optimization. The refinement simply embeds `ind` into the non negative integers as follows:

```
(declare-refinement ind => int
  ()                          ; unparameterized
  "ind" "integer"             ; src and target types
  ()                          ; no state
  ([i "ind"])                 ; formal variable decls
  ([i* "(ind=>int i)"])       ; bindings
  (["(inc i)" "(+ 1 i*)"]     ; implementation of inc
   ["(dec i)" "(sel (= 0 i*) (- i* 1) 0)"] ; of dec
   ["0_ind" "0"])             ; implementation of 0_ind
  ()
  (['ind=>int:inv "(ind<=int i*)" "i"]))
```

Inputs: (instr, a)
Outputs: (res)

ser	(inst-cat instr)	ptr*:Seq.	s*:Seq.	tl*:Seq.	mem:Seq.	res:Seq.	x:Seq.
#	psh-op	(+ 1 ptr*)	(ind<=int ptr*)	s*	(wr mem (ind<=int ptr*) [a s*])	a	#
#	drp-op	ptr*	tl*	(2nd (rd mem tl*))	mem	(1st (rd mem tl*))	#
0	alu-op	ptr*	s*	tl*	mem	#	res
1	alu-op	ptr*	s*	(2nd (rd mem tl*))	mem	(1st (rd mem tl*))	x
2	alu-op	ptr*	s*	tl*	mem	(alu (inst->op instr) x res)	#
3	alu-op	(+ 1 ptr*)	(ind<=int ptr*)	tl*	(wr mem (ind<=int ptr*) [res tl*])	res	#

The above table shows the result of applying this refinement to `ptr`. The new signal, `ptr*`, is an `integer`. Since this is a local refinement, consumers of the original `ptr` now accept `(ind<=int ptr*)` in its place. At a low level, this type coercion may simply be projection out of a bit vector; e.g., the address space is 24-bit while the integers are 32-bit. Thus the overhead of coercion need not be high.

Step 6

Inputs: (instr, a)
Outputs: (res)

ser	(inst-cat instr)	ptr*:Seq.	s*:Seq.	tl*:Seq.	mem:Seq.	res:Seq.	x:Seq.	cell:Comb.
#	psh-op	(alu 0b00 1 ptr*)	(ind<=int ptr*)	s*	(wr mem (ind<=int ptr*) [a s*])	a	#	#
#	drp-op	ptr*	tl*	(2nd cell)	mem	(1st cell)	#	(rd mem tl*)
0	alu-op	ptr*	s*	tl*	mem	#	res	#
1	alu-op	ptr*	s*	(2nd cell)	mem	(1st cell)	x	(rd mem tl*)
2	alu-op	ptr*	s*	tl*	mem	(alu (inst->op instr) x res)	#	#
3	alu-op	(alu 0b00 1 ptr*)	(ind<=int ptr*)	tl*	(wr mem (ind<=int ptr*) [res tl*])	res	#	#

To prepare for memory factorization the above table introduces a combinational signal `cell` to hold the common subterm `(rd mem tl*)` for actions `[# drp-op] [1 alu-op]` in signals `tl*` and `res`. Also, note the rewriting of `+` applications as instances of `alu` in anticipation of ALU factorization. The signals `mem` and `cell` are the subjects of memory factorization. Uniform signal and function usage reduces the interconnects and instructions resulting from factorization.

Step 7

The next derivation subgoal is to produce uniform arguments for applications of `wr` in signal `mem` by serializing `psh-op`.

ptr*	s*	tl*	mem	res	x
ptr*	(ind<=int ptr*)	s*	mem	a	1
(alu 0b00 x ptr*)	s*	tl*	(wr mem (ind<=int ptr*) [res tl*])	res	#
#	#	#	#	#	#

ptr*	s*	tl*	mem	res	x
ptr*	(ind<=int ptr*)	s*	mem	a	1
(alu 0b00 1 ptr*)	(ind<=int ptr*)	s*	(wr mem (ind<=int ptr*) [a s*])	a	#
(alu 0b00 1 ptr*)	(ind<=int ptr*)	s*	(wr mem (ind<=int ptr*) [a s*])	a	#

⇓

Inputs: (instr, a)
Outputs: (res)

ser	(inst-cat instr)	ptr*:Seq.	s*:Seq.	tl*:Seq.	mem:Seq.	res:Seq.	x:Seq.	cell:Comb.
0	psh-op	ptr*	(ind<=int ptr*)	s*	mem	a	1	#
1	psh-op	(alu 0b00 x ptr*)	s*	tl*	(wr mem (ind<=int ptr*) [res tl*])	res	#	#
#	drp-op	ptr*	tl*	(2nd cell)	mem	(1st cell)	#	(rd mem tl*)
0	alu-op	ptr*	s*	tl*	mem	#	res	#
1	alu-op	ptr*	s*	(2nd cell)	mem	(1st cell)	x	(rd mem tl*)
2	alu-op	ptr*	s*	tl*	mem	(alu (inst->op instr) x res)	#	#
3	alu-op	(alu 0b00 1 ptr*)	(ind<=int ptr*)	tl*	(wr mem (ind<=int ptr*) [res tl*])	res	#	#

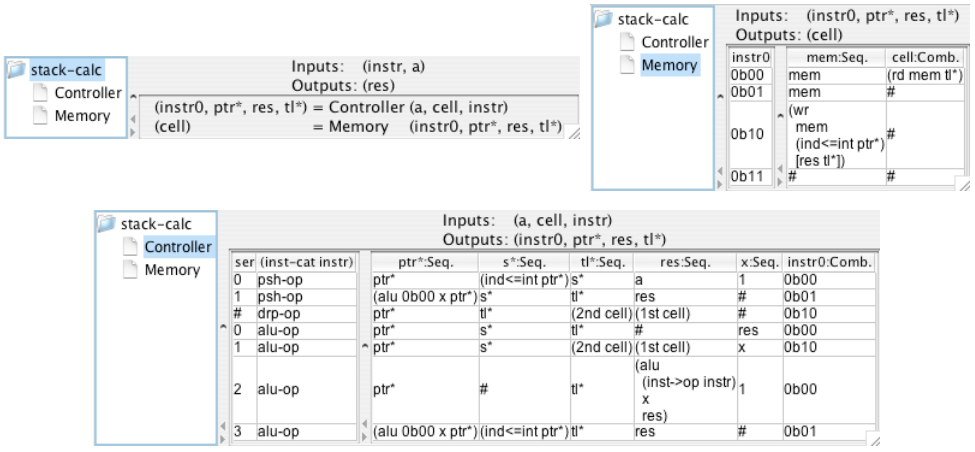
⇓

Inputs: (instr, a)
Outputs: (res)

ser	(inst-cat instr)	ptr*:Seq.	s*:Seq.	tl*:Seq.	mem:Seq.	res:Seq.	x:Seq.	cell:Comb.
0	psh-op	ptr*	(ind<=int ptr*)	s*	mem	a	1	#
1	psh-op	(alu 0b00 x ptr*)	s*	tl*	(wr mem (ind<=int ptr*) [res tl*])	res	#	#
#	drp-op	ptr*	tl*	(2nd cell)	mem	(1st cell)	#	(rd mem tl*)
0	alu-op	ptr*	s*	tl*	mem	#	res	#
1	alu-op	ptr*	s*	(2nd cell)	mem	(1st cell)	x	(rd mem tl*)
2	alu-op	ptr*	#	tl*	mem	(alu (inst->op instr) x res)	1	#
3	alu-op	(alu 0b00 x ptr*)	(ind<=int ptr*)	tl*	(wr mem (ind<=int ptr*) [res tl*])	res	#	#

The top table (above) shows the two step serialization proposal for the register actions guarded by `psh-op`. The first step stores `a` in `res` and `s*` in `tl*`. The subsequent call to `wr` now operates on `[res tl*]` as its companion does in the action guarded by `[3 alu-op]` (middle table). Also, the first step places the constant `1` into the temprary register `x` to anticipate a more uniform application of `alu` in the next step. Finally, this `alu` *homogenization* extends to a reserialization of `alu-op`, steps 2 & 3 (bottom table); as before, the temporary register `x` holds the constant `1`. As a result all three `alu` applications carry `x` in the second argument.

Step 8



The above system of tables decompose the stack calculator into a memory and a controller. This decomposition cuts across the signals that implement abstract stacks: the free cell pointer `ptr*`, and stack reference signals `s*` and `tl*` remain in the controller, while the abstract memory becomes a separate component.

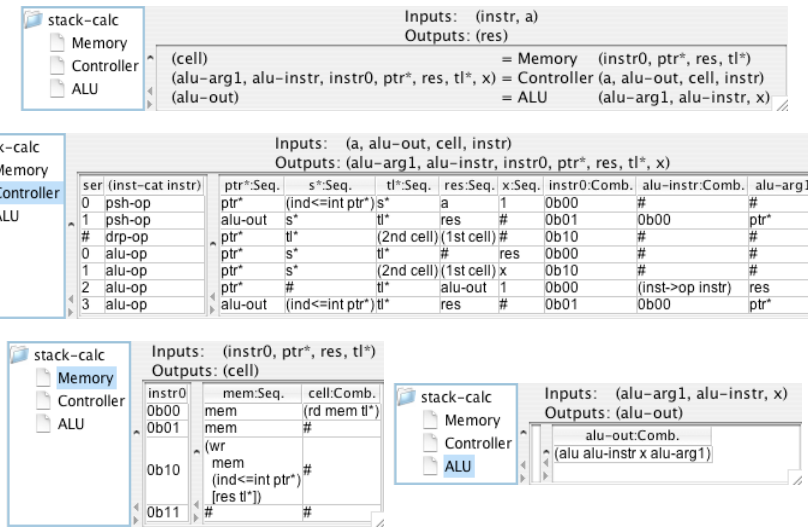
Step 9

Inputs: (a, cell, instr)
Outputs: (instr0, ptr*, res, tl*)

ser	(inst-cat instr)	ptr*:Seq.	s*:Seq.	tl*:Seq.	res:Seq.	x:Seq.	instr0:Comb.	alu-out:Comb.	alu-instr:Comb.	alu-arg1:Comb.
0	psh-op	ptr*	(ind<=int ptr*)	s*	a	1	0b00	#	#	#
1	psh-op	alu-out	s*	tl*	res	#	0b01	(alu alu-instr x alu-arg1)	0b00	ptr*
#	drp-op	ptr*	tl*	(2nd cell)	(1st cell)	#	0b10	#	#	#
0	alu-op	ptr*	s*	tl*	#	res	0b00	#	#	#
1	alu-op	ptr*	s*	(2nd cell)	(1st cell)	x	0b10	#	#	#
2	alu-op	ptr*	#	tl*	alu-out	1	0b00	(alu alu-instr x alu-arg1)	(inst->op instr)	res
3	alu-op	alu-out	(ind<=int ptr*)	tl*	res	#	0b01	(alu alu-instr x alu-arg1)	0b00	ptr*

The table above expands applications of `alu` and their arguments into combinational host signals. The combinational signals `alu-instr` and `alu-arg1` leverage the decision table guards to switch the inputs to `alu`. Since `x` is the second argument for every application, there is no need to provide a separate argument host signal. Note that rescheduling `alu-op` or `psh-op` could produce a completely uniform application of `alu` (as with `wr` and `rd` in the memory factorization) at the cost of some extra register transfers and a temporary alu-decode register for the instructions. As with the coercions from `integer` to `ind`, the "alu-decode" (`inst->op`) may be as simple as projecting the right bits out of the instruction's bit vector.

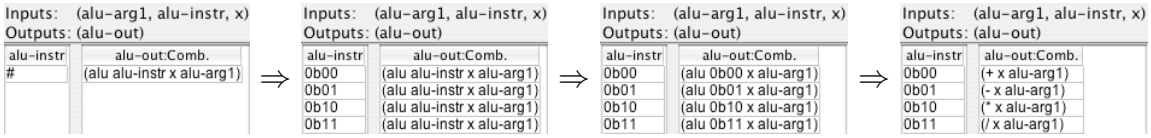
Step 10



Factoring the `alu` produces the above system.

Step 11

The derivation finishes by expanding the ALU component (below). The input `alu--instr` is the soul gaurd (first and second tables). Once the guard identities are instantiated (third table), the alu identities recover the four basic operations (final table).



Inputs: (alu-arg1, alu-instr, x)
Outputs: (alu-out)

alu-instr	alu-out:Comb.
#	(alu alu-instr x alu-arg1)

⇒

Inputs: (alu-arg1, alu-instr, x)
Outputs: (alu-out)

alu-instr	alu-out:Comb.
0b00	(alu alu-instr x alu-arg1)
0b01	(alu alu-instr x alu-arg1)
0b10	(alu alu-instr x alu-arg1)
0b11	(alu alu-instr x alu-arg1)

⇒

Inputs: (alu-arg1, alu-instr, x)
Outputs: (alu-out)

alu-instr	alu-out:Comb.
0b00	(alu 0b00 x alu-arg1)
0b01	(alu 0b01 x alu-arg1)
0b10	(alu 0b10 x alu-arg1)
0b11	(alu 0b11 x alu-arg1)

⇒

Inputs: (alu-arg1, alu-instr, x)
Outputs: (alu-out)

alu-instr	alu-out:Comb.
0b00	(+ x alu-arg1)
0b01	(- x alu-arg1)
0b10	(* x alu-arg1)
0b11	(/ x alu-arg1)

Summary

This derivation develops high level architecture from an architecture agnostic specification. While the specification has just one abstractly typed register to hold the stack, the resulting architecture uses reference and integer registers to hold the current stack reference `s*`, the popped stack reference `tl*`, the stack top `res`, reference, the next free cell reference `ptr*`, and a temporary register `x`. Each of the three components plays a role in stack implementation. The controller holds all of the reference registers, and delegates updates to the ALU and memory. The memory stores and retrieves the stack content. The ALU computes the increment for the next-free-cell pointer as well as the arithmetic demands of the calculator's user - a nice reuse of necessary architecture.

Chapter 9

Case Studies

9.1 SchemEngine Garbage Collector

This study factors a heap garbage collector; the example first appears in [96]. Wehrmeister's LISP virtual machine derivation [103] used a primitive version of this garbage collector. Later, Burger enhanced the specification into the form presented here for his Scheme-machine derivation [16]. Both derivations resulted in working hardware. The factorization shows the feasibility of behavior-table-oriented derivation on real-world designs.

This specification follows a functional modeling approach and represents memory with abstract data-types. The stop-and-copy algorithm literally "swaps" the two memory signals, `new` and `old`, the end of collection. The derivation factors the abstract memory signals and operations from the control algorithm, and uses data refinement to implement memory swapping with a switch that indicates which of two memories is *new*. The resulting memory component is well suited for implementation with two single-ported random-access memories and a switch flag or XOR-gate. The controller's remaining term functions either increment values, add two values, mask bit-vectors or embed bit-vectors. Subsequent factorization on

the add/increment/decrement functions (not performed here) would produce a pure register-transfer controller.

9.1.1 Specification

The garbage collector specification uses two enumerated types: one for control state and another for classifying of heap elements. 32-bit vectors encode heap elements, while 24-bit vectors define memory addresses. Memory is a parameterized type over *address* and *data*, instantiated to 24-bit and 32-bit vectors respectively. Garbage collection follows a modified stop-and-copy approach.

The specification uses two sequential signals `old` and `new` to represent streams of memory half-spaces. After all reachable heap elements are copied from `old` to `new`, the control state preceding transition into `idle` swaps the two abstract memory values; see signals `old` and `new` in the action table of Figure 18 at the row guarded by (`driver`, `#`, `true`, `#`, ..., `#`). The `scan` register holds the address of the heap value in `new` whose children are to be copied over from `old`. The `next` register holds the address of next available free cell in `new`. The collection loop ends with the convergence of `scan` and `next`. Register `H` holds the word (in `new`) indexed by `scan`. When `H` is a pointer type (i.e., not an immediate value), `D` is the word (in `old`) that `H` references. `C` is a count register for copying contiguous cell blocks (as with a vector or a fixed length object). Finally, the `ak` signal indicates when collection is finished.

```
(define-enum-alg gc-state
  (idle driver next-obj obj-type vec copy) () () ())

(define-enum-alg type-class
  (fixed vector byte-vec) () () ())

(define-param-alg memory
  (addr data)
  ()
  ((wr (memory addr data) memory)
   (rd (memory addr) data))
  ((m memory) (a addr) (d data) (e data))
  (('mem1 (rd (wr m a d) a) d)
   ('mem2 (wr m a (rd m a)) m)
   ('mem3 (wr (wr m a d) a e) (wr m a e))))

(declare-funcs gc-help
  () ;; no parameters
  ((+val ("bvec{24}" "bvec{24}") "bvec{24}")
   (+1val ("bvec{24}") "bvec{24}")
   (-1val ("bvec{24}") "bvec{24}")
   (+1cell ("bvec{32}") "bvec{32}")
   (=val ("bvec{24}" "bvec{24}") "boolean")
   (pointer? ("bvec{32}") "boolean")
   (bv-head? ("bvec{32}") "boolean")
   (fw-tag? ("bvec{32}") "boolean")
   (tclass ("bvec{32}") "type-class")
   (cell->val ("bvec{32}") "bvec{24}")
   (b->w ("bvec{24}") "bvec{24}")
   (size ("bvec{32}") "bvec{24}")
   (mk-cell-1 ("bvec{32}" "bvec{32}") "bvec{32}")
   (mk-cell-2 ("bvec{32}" "bvec{24}") "bvec{32}")
   (mk-fw-cell ("bvec{24}") "bvec{32}")
    ) () ())
```

Figure 17: Datatype declarations for GC

Inputs: (rq)
Outputs: (ak)

now	rq	(=val scan next)	(pointer? h)	(bv-head? h)	(fw-tag? d)	(tclass h)	(=val c 0x000000)	(=val c 0x000001)
idle	true	#	#	#	#	#	#	#
idle	false	#	#	#	#	#	#	#
driver	#	true	#	#	#	#	#	#
driver	#	false	#	#	#	#	#	#
next-obj	#	#	true	#	#	#	#	#
next-obj	#	#	false	true	#	#	#	#
next-obj	#	#	false	false	#	#	#	#
obj-type	#	#	#	#	true	#	#	#
obj-type	#	#	#	#	false	fixed	#	#
obj-type	#	#	#	#	false	vector	#	#
obj-type	#	#	#	#	false	byte-vec	#	#
vec	#	#	#	#	#	#	false	#
vec	#	#	#	#	#	#	true	#
copy	#	#	#	#	#	#	#	false
copy	#	#	#	#	#	#	#	true

now:Seq.	old:Seq.	new:Seq.	h:Seq.	d:Seq.	c:Seq.	scan:Seq.	next:Seq.	ak:Seq.
next-obj	old	new	0x50000000	d	c	0x000000	next	false
idle	old	new	h	#	#	#	#	true
idle	new	old	h	d	c	scan	next	true
next-obj	old	new	(rd new scan)	d	c	scan	next	false
obj-type	old	(wr new scan (mk-cell-2 h next))	h	(rd old (cell->val h))	c	scan	next	false
driver	old	new	h	d	c	(+1val (+val scan (b->w (cell->val h))))	next	false
driver	old	new	h	d	c	(+1val scan)	next	false
driver	old	(wr new scan (mk-cell-1 h d))	h	d	c	(+1val scan)	next	false
copy	(wr old (cell->val h) (mk-fw-cell next))	new	(+1cell h)	d	(size h)	scan	next	false
vec	(wr old (cell->val h) (mk-fw-cell next))	new	(+1cell h)	d	(cell->val d)	scan	next	false
vec	(wr old (cell->val h) (mk-fw-cell next))	new	(+1cell h)	d	(b->w (cell->val d))	scan	next	false
driver	old	(wr new next d)	h	d	c	(+1val scan)	(+1val next)	false
copy	old	(wr new next d)	(+1cell h)	(rd old (cell->val h))	c	scan	(+1val next)	false
driver	old	(wr new next d)	h	d	c	(+1val scan)	(+1val next)	false
copy	old	(wr new next d)	(+1cell h)	(rd old (cell->val h))	(-1val c)	scan	(+1val next)	false

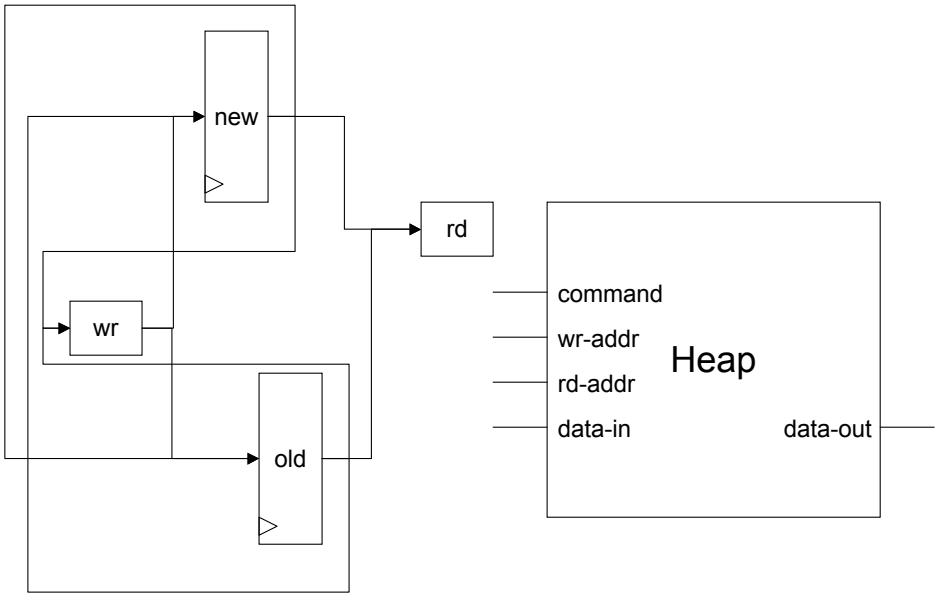
⇒

now:Seq.	old:Seq.	new:Seq.	h:Seq.	d:Seq.	app0:Comb.	app1:Comb.	arg0:Comb.	arg1:Comb.	arg2:Comb.
next-obj	old	new	0x50000000	d	#	#	#	#	#
idle	old	new	h	#	#	#	#	#	#
idle	new	old	h	d	#	#	#	#	#
next-obj	old	new	app0	d	(rd new arg0)	#	scan	#	#
obj-type	old	app1	h	app0	(rd old arg0)	(wr new arg2 arg1)	(cell->val h)	(mk-cell-2 h next)	scan
driver	old	new	h	d	#	#	#	#	#
driver	old	new	h	d	#	#	#	#	#
driver	old	app1	h	d	#	(wr new arg0 arg1)	scan	(mk-cell-1 h d)	#
copy	app1	new	(+1cell h)	d	#	(wr old arg0 arg1)	(cell->val h)	(mk-fw-cell next)	#
vec	app1	new	(+1cell h)	d	#	(wr old arg0 arg1)	(cell->val h)	(mk-fw-cell next)	#
vec	app1	new	(+1cell h)	d	#	(wr old arg0 arg1)	(cell->val h)	(mk-fw-cell next)	#
driver	old	app1	h	d	#	(wr new arg0 arg1)	next	d	#
copy	old	app1	(+1cell h)	app0	(rd old arg0)	(wr new arg2 arg1)	(cell->val h)	d	next
driver	old	app1	h	d	#	(wr new arg0 arg1)	next	d	#
copy	old	app1	(+1cell h)	app0	(rd old arg0)	(wr new arg2 arg1)	(cell->val h)	d	next

⇒

app0:Comb.	app1:Comb.	arg0:Comb.	arg1:Comb.	arg2:Comb.
#	#	#	#	#
#	#	#	#	#
#	#	#	#	#
(rd new arg0)	#	scan	#	#
(rd old arg0)	(wr new arg2 arg1)	(cell->val h)	(mk-cell-2 h next)	scan
#	#	#	#	#
#	#	#	#	#
#	(wr new arg2 arg1)	#	(mk-cell-1 h d)	scan
#	(wr old arg2 arg1)	#	(mk-fw-cell next)	(cell->val h)
#	(wr old arg2 arg1)	#	(mk-fw-cell next)	(cell->val h)
#	(wr old arg2 arg1)	#	(mk-fw-cell next)	(cell->val h)
#	(wr new arg2 arg1)	#	d	next
(rd old arg0)	(wr new arg2 arg1)	(cell->val h)	d	next
#	(wr new arg2 arg1)	#	d	next
(rd old arg0)	(wr new arg2 arg1)	(cell->val h)	d	next

Figure 18: GC's behavior table specification (top); result of expanding `wr`, `rd`, and their arguments into combinational signals (bottom left); result of permuting `argX` signals for uniformity (bottom right). In both bottom tables, the view is limited to the signals that changed; the decision table is the same throughout.



gc
GC Control
Dual Ported Memory

Inputs: (rq)
Outputs: (ak)

(ak, arg0, arg1, arg2, instr0) = GC Control (app0, rq)
(app0) = Dual Ported Memory (arg0, arg1, arg2, instr0)

gc
GC Control
Dual Ported Memory

Inputs: (arg0, arg1, arg2, instr0)
Outputs: (app0)

instr0	old:Seq.	new:Seq.	app0:Comb.	app1:Comb.
00	new	old	#	#
01	old	new	(rd new arg0)	#
02	old	new	#	#
03	app1	new	#	(wr old arg2 arg1)
04	old	app1	#	(wr new arg2 arg1)
05	old	app1	(rd old arg0)	(wr new arg2 arg1)
06	#	#	#	#
07	#	#	#	#

Inputs: (app0, rq)
Outputs: (ak, arg0, arg1, arg2, instr0)

now:Seq.	h:Seq.	d:Seq.	c:Seq.	scan:Seq.	next:Seq.	ak:Seq.	arg0:Comb.	arg1:Comb.	arg2:Comb.	instr0:Comb.
next-obj	0x50000000	d	c	0x000000	next	false	#	#	#	00
idle	h	#	#	#	#	true	#	#	#	00
idle	h	d	c	scan	next	true	#	#	#	01
next-obj	app0	d	c	scan	next	false	scan	#	#	02
obj-type	h	app0	c	scan	next	false	(cell->val h)	(mk-cell-2 h next)	scan	03
driver	h	d	c	(+1val (+val scan (b->w (cell->val h))))	next	false	#	#	#	00
driver	h	d	c	(+1val scan)	next	false	#	#	#	00
driver	h	d	c	(+1val scan)	next	false	#	(mk-cell-1 h d)	scan	04
copy	(+1cell h)	d	(size h)	scan	next	false	#	(mk-fw-cell next)	(cell->val h)	05
vec	(+1cell h)	d	(cell->val d)	scan	next	false	#	(mk-fw-cell next)	(cell->val h)	05
vec	(+1cell h)	d	(b->w (cell->val d))	scan	next	false	#	(mk-fw-cell next)	(cell->val h)	05
driver	h	d	c	(+1val scan)	(+1val next)	false	#	d	next	04
copy	(+1cell h)	app0	c	scan	(+1val next)	false	(cell->val h)	d	next	03
driver	h	d	c	(+1val scan)	(+1val next)	false	#	d	next	04
copy	(+1cell h)	app0	(-1val c)	scan	(+1val next)	false	(cell->val h)	d	next	03

Figure 19: The GC specification uses abstract memory signals and operations; factoring them from the controller encapsulates the abstraction with a separate process. Details are shown with the tabular representation.

9.1.2 Factorization

This process depends on the heuristically guided factorization assistants, `expand-apps`, `permute-comb-signals`, and `factor-signals`, explained in Chapter 6.4. The garbage collector specification uses two abstract memories as internal signals. The first goal is to decompose this specification into a controller that interacts with an external memory. In addition to isolating the memory signals themselves, it is further necessary to separate its accessors; i.e., the `rd` function. The first step is to expand applications of `rd` and `wr` into host signals using `expand-apps`. This tactical transformation automatically creates combinational signals `app0` and `app1`, instantiates them to `rd` and `wr` respectively, and places their arguments into new combinational signals `arg0`, `arg1` and `arg2`. The automatic argument allocation lacks uniformity, with `rd` taking `arg2` as its input in some places but `arg0` in other places. Another convenience transformation `permute-comb-signals` permutes actions among same-typed combinational signals and updates their client terms. It homogenizes parameter content for `arg0` (address input to `rd`), `arg1` (data input to `wr`) and `arg2` (address input to `wr`). Argument permutation reduces the number of instructions inside the factored component; `(wr old arg1 arg0)` and `(wr old arg1 arg2)` would generate different commands. Applying `factor-signals` to the target signals `old`, `new`, `app0` and `app1` extracts distinct actions on the four signals, assigning each action to a unique bit-vector code. This indexed table of actions forms the behavior table for the abstract memory. A combinational signal, which specifies instructions to the abstract memory, replaces the four memory signals from controller. Figure 19 shows the result of this factorization.

9.1.3 Data Refinement

The next step eliminates explicit memory swapping (instruction `00`) in favor of state tracking. Let `gcm` be the tuple `[m1 m2]`. A collection of wrapper functions (defined in the code below) `wr-new`, `wr-old`, `rd-new`, `rd-old`, and `swap`, packages the actions on individual signals `m1` and `m2` as operations the tuple `gcm`. The `old` and `new` versions access the first and second tuple components respectively, while swap exchanges their values.

```
;; Formals
((m1 "memory{bvec{24} bvec{32}}")
 (m2 "memory{bvec{24} bvec{32}}")
 (a "bvec{24}")
 (d "bvec{32}"))

;; Identities
 ('wr-old-def "(wr-old [m1 m2] a d)" "[(wr m1 a d) m2]")
 ('wr-new-def "(wr-new [m1 m2] a d)" "[m1 (wr m2 a d)]")
 ('rd-old-def "(rd-old [m1 m2] a)"   "(rd m1 a)")
 ('rd-new-def "(rd-new [m1 m2] a)"   "(rd m2 a)")
 ('swap-def   "(swap [m1 m2])"       "[m2 m1]")
```

The following data refinement represents this bundled type as a triple with two memories and a boolean switch. The switch determines which memory to access, and `swap` exchanges memory half-spaces by inverting the switch. The identities below show how the wrapper functions map to switch-guarded read and write accesses.

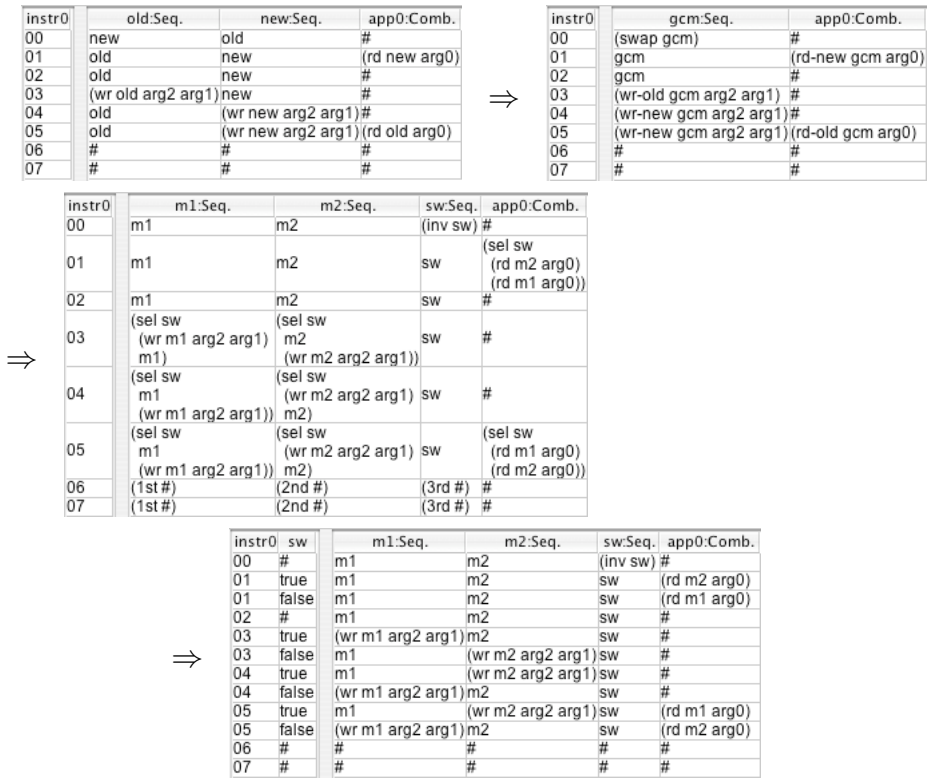


Figure 20: Elimination of memory swapping by switch refinement

```
;; Formals
([gcm "[memory{bvec{24} bvec{32}} memory{bvec{24} bvec{32}}]"]
 [a "bvec{24}"]
 [w "bvec{32}"])

;; Bindings
([m1 "(1st (2mem=>bit gcm))"]
 [m2 "(2nd (2mem=>bit gcm))"]
 [sw "(3rd (2mem=>bit gcm))"])

;; Refinement mappings
 ["(wr-old gcm a w)" "[(sel sw (wr m1 a w) m1)
                       (sel sw m2 (wr m2 a w)) sw]"]
 ["(wr-new gcm a w)" "[(sel sw m1 (wr m1 a w))
                       (sel sw (wr m2 a w) m2) sw]"]
 ["(swap gcm)"       "[m1 m2 (inv sw)]"]
 ["(rd-old gcm a)"   "(sel sw (rd m1 a) (rd m2 a))"]
 ["(rd-new gcm a)"   "(sel sw (rd m2 a) (rd m1 a))"]
```

Figure 20 shows the three steps in the switch derivation: The first step bundles the two memories into one value `gcm` (garbage collected memory). The second applies the switch-guarded refinement to `gcm`, and splits the implementing tuple into `m1`, `m2`, and `sw`. The last step expands the switch into the decision table.

9.2 The SECD Machine

SECD is an abstract machine that defines operational semantics for LISP[63]. The name comes from its four "registers"—*stack, environment, control, dump*—which use abstract pairs to hold an argument stack, a binding environment, the nested byte code program, and a function call stack (dump). LISP machines have been the subject of many formal verification and design efforts, including the VLISP project [35], a SECD hardware verification in HOL [6], and the SchemEngine derivation [16, 56]. Independently of VLISP and the HOL verification of SECD, Wehrmeister derived a hardware implementation for a SECD specification with DDD [103]. Wehrmeister's starting specification is far more detailed than any machine used for operational semantics. The purpose of this study is to take a high-level SECD specification and derive a good approximation of Wehrmeister's starting point.

9.2.1 SECD machine specification

The SECD machine interprets structured lists of byte-codes (control). Primitive functions and control flow operations have their own byte-code, while user-defined functions are lists of byte-codes. The machine adheres to the following invariant: Let

```
(define secd
 (lambda(exp)
  (letrec
   ((exec
     (lambda (s e c d)
      (case (car c)
       ('RTN  (exec (cons (car s)(car d)) (cadr d) (caddr d) (cdddr d)))
       ('DUM  (exec s (cons () e) (cdr c) d))
       ('AP   (exec nil (cons (cadr s)(cdar s))
                (caar s) (cons (cddr s)(cons e (cons (cdr c) d)))))
       ('RAP  (exec nil (set-car! (cdar s)(cadr s)) (caar s)
                (cons (cddr s)(cons (cdr e)(cons (cdr c) d)))))
       ('SEL  (exec (cdr s) e (if (car s)(cadr c)(caddr c))
                (cons (cdddr c) d)))
       ('JOIN (exec s e (car d) (cdr d)))
       ('CAR  (exec (cons (caar s)(cdr s)) e (cdr c) d))
       ('CDR  (exec (cons (cdar s)(cdr s)) e (cdr c) d))
       ('CONS (exec (cons (cons (car s)(cadr s))(cddr s)) e (cdr c) d))
       ('LD   (exec (cons (locate (cadr c) e) s) e (cddr c) d))
       ('LDC  (exec (cons (cadr c) s) e (cddr c) d))
       ('LDF  (exec (cons (cons (cadr c) e) s) e (cddr c) d))
       ('ATOM (exec (cons (atom? (car s))(cdr s)) e (cdr c) d))
       ('EQ   (exec (cons (eq? (car s)(cadr s))(cddr s)) e (cdr c) d))
       ('LEQ  (exec (cons (<=? (car s)(cadr s))(cddr s)) e (cdr c) d))
       ('ADD  (exec (cons (+ (car s)(cadr s))(cddr s)) e (cdr c) d))
       ('SUB  (exec (cons (- (car s)(cadr s))(cddr s)) e (cdr c) d))
       ('STOP (exec s e c d))
       (else  (exec s e c d))))))
   (exec nil nil exp nil))))
```

Figure 21: A high-level SECD s-expression specification (HS) to which Wehrmeister attributes the design intent of his machine.

the control register hold a valid program in its initial segment. After execution of the program, the machine state has the same environment and dump prior to execution, and the stack is expanded by one to include the result of the program, and the control register's initial segment has been removed. Figure 21 shows a high-level specification of SECD. This code is a variant of the Henderson's formulation of SECD [42] for the operational semantics of LISP.

While the SECD machine gives a more detailed semantics than the interpreter based semantics, still has ambiguities. `rplaca` is a pseudofunction which is evaluated for its effect and not its value. To functionally specify the action of `rplaca`, one must define its "effects" and interplay with `cons`, `car`, and `cdr` over a more concrete data type that fully exposes the "effects" in their domain and range. Understanding the SECD machine depends upon its core data type, mutable s-expressions.

Because the construction of circular lists with side-effects can not be characterized algebraically, I have chosen to drop the `rap` and `set` instructions from the high-level specification. These would be added to the specification after refining s-expressions into memory accesses (`alloc`, `setcar!`, `setcdr!`).

While the high-level specification can draw its atoms from an infinite symbol space, implementations need to precisely define this data-type. Wehrmeister's specification supports characters and symbols. Characters are an atomic data-type, but symbols are implemented as lists of characters where the first pair bears the "symbol" tag rather than the "pair" tag. Thus, our derivations's starting point adds characters to the atoms as well as primitives that transform symbols to lists of characters and vice versa. Numbers are ultimately bounded bit-vector representations, but this derivation

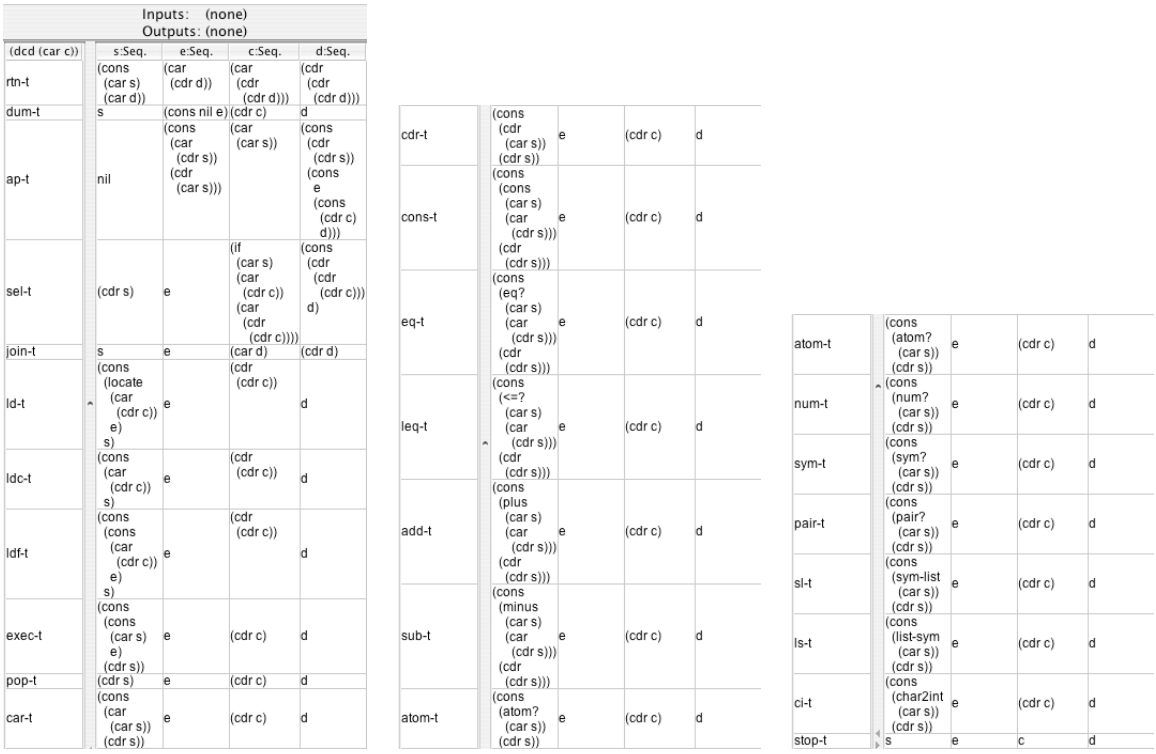
treats them as ideal mathematical entities throughout.

Because of the necessary side-effects in the high-level specification, Wehrmeister chose to specify over a heap data-type. This eliminated the need for side-effects, but resulted in a far more detailed specification. His specification serializes access to the heap and contains some non-trivial retimings. Moreover, the specification interfaces with a garbage collector and processes input and output. For the remainder of this chapter, *WS* refers to the specification of Wehrmeister's SECD machine, while *HS* refers to the high-level operational semantics specification.

Our approximation of the WS does not include the garbage collector interface or the I/O handling. Like the side-effect necessary for the recursive apply operation (`rap`), these features may be added on to the result of our derivation. The study justifies transition from nested abstract pairs to references over a heap of cells, WS's particular scheduling of intermediate operations, and the storage of intermediate results in new registers. The initial behavior table (Figure 22) for this derivation is a by-hand translation of the s-expression in Figure 21.

9.2.2 Specification Signature

The core type declaration is `lt`, for *LISP type*. The signature's intended model is the collection of atoms and pairs over model elements. The declaration does not prescribe particular atoms or the behavior of most of its functions, but mandates some minimal well-formedness conditions (i.e., function signatures) and identities for the term's equational logic. The constants `nil`, `tru` and `fls`, represent the LISP



Inputs: (none)
Outputs: (none)

(dcd (car c))	s:Seq.	e:Seq.	c:Seq.	d:Seq.
rtn-t	(cons (car s) (car d))	(car (cdr d))	(car (cdr (cdr d)))	(cdr (cdr (cdr d)))
dum-t	s	(cons nil e)	(cdr c)	d
ap-t	nil	(cons (car (cdr s)) (cdr (car s)))	(car (car s))	(cons (cdr (cdr s)) (cons e (cons (cdr c) d)))
sel-t	(cdr s)	e	(if (car s) (car (cdr c)) (car (cdr (cdr c))))	(cons (cdr (cdr (cdr c))) d)
join-t	s	e	(car d)	(cdr d)
ld-t	(cons (locate (car (cdr c)) e) s)	e	(cdr (cdr c))	d
ldc-t	(cons (car (cdr c)) s)	e	(cdr (cdr c))	d
ldf-t	(cons (cons (car (cdr c)) e) s)	e	(cdr (cdr c))	d
exec-t	(cons (cons (car s) e) (cdr s))	e	(cdr c)	d
pop-t	(cdr s)	e	(cdr c)	d
car-t	(cons (car (car s)) (cdr s))	e	(cdr c)	d

cdr-t	(cons (cdr (car s)) (cdr s))	e	(cdr c)	d
cons-t	(cons (cons (car s) (car (cdr s))) (cdr (cdr s)))	e	(cdr c)	d
eq-t	(cons (eq? (car s) (car (cdr s))) (cdr (cdr s)))	e	(cdr c)	d
leq-t	(cons (<=? (car s) (car (cdr s))) (cdr (cdr s)))	e	(cdr c)	d
add-t	(cons (plus (car s) (car (cdr s))) (cdr (cdr s)))	e	(cdr c)	d
sub-t	(cons (minus (car s) (car (cdr s))) (cdr (cdr s)))	e	(cdr c)	d
atom-t	(cons (atom? (car s)) (cdr s))	e	(cdr c)	d

atom-t	(cons (atom? (car s)) (cdr s))	e	(cdr c)	d
num-t	(cons (num? (car s)) (cdr s))	e	(cdr c)	d
sym-t	(cons (sym? (car s)) (cdr s))	e	(cdr c)	d
pair-t	(cons (pair? (car s)) (cdr s))	e	(cdr c)	d
sl-t	(cons (sym-list (car s)) (cdr s))	e	(cdr c)	d
ls-t	(cons (list-sym (car s)) (cdr s))	e	(cdr c)	d
ci-t	(cons (char2int (car s)) (cdr s))	e	(cdr c)	d
stop-t	s	e	c	d

Figure 22: SECD behavior table specification

empty-list and boolean values. Functions `cons`, `car` and `cdr` are a binary constructor with accessors as shown by the `access1` and `access2` identities. The functions `eq?`, `<=?`, `plus`, `minus`, `nil?`, `atom?`, `num?`, `sym?`, and `pair?` have the standard interpretations in the intended model, but are unconstrained by the identities because the derivation does not reason about the application of these functions. Similarly, the functions `locate`, `sym-list`, `list-sym`, and `char2int`, are not explicitly rewritten in the derivation and consequently are not accompanied by identities. However, their anticipated implementation is unclear. They will be factored out in the end result. WS expresses the predicates `atom?`, `num?`, `sym?`, and `pair?` with a parameterized function `test?` that specifies the desired predicate with input instructions;

the declarative block `secd-helpers` makes this connection explicit. The enumerated type declaration `secd-state` behaves as a label-set that corresponds to decoded byte-codes from the control signal, `c`.

Since `lt` can only declare signatures and identities over type `lt`, the `declare-funcs` block `secd-helpers` completes the identities over the specification types. `dcd` takes a `lt` and produces an instruction token in `secd-state`. This is necessary to mimic the specification's `case` structure. The declarative block also registers predicate identities for `test?`. The function `test-inst?` defines the behavior for `test?` when its first agrument decodes to `atom-t`, `num-t`, `sym-t`, or `pair-t`. `true?` maps `lt`s to booleans so that `(if k a b)` may be defined by `(sel (true? k) a b)`.

9.2.3 Derivation Strategy

The high-level derivation strategy consists of five steps: introducing fetch register and update, rewriting function applications in the high-level specification to match those in WS, serializing the memory access operations `cons`, `car` and `cdr`, refining HS's `lt` data-type into WS's heap (`lmem`) over cells (`lc`), and rescheduling the implementation of `cons` (a term that entails three memory accesses). The serialize/refine-data/reserialize order of tasks prevents the excessive term size resulting from serial-store access imposed by stateful data refinement. Had data refinement preceded serialization, the terms would have become impractically large as in Figure 24.

```
(define-enum-alg secd-state
  (rtn-t dum-t ap-t sel-t join-t car-t cdr-t cons-t ld-t
    ldc-t ldf-t atom-t eq-t leq-t add-t sub-t sl-t ls-t
    ci-t num-t sym-t pair-t exec-t pop-t stop-t) () () ())

(define-term-alg lt
  ;; Constants
  (nil err tru fls)
  ;; Functions with arity
  ((cons 2) (car 1) (cdr 1)
   (nil? 1) (atom? 1) (num? 1) (pair? 1) (sym? 1)(eq? 2)
   (test? 2)
   (<=? 2) (plus 2) (minus 2)
   (if 3)
   (sym-list 1) (list-sym 1) (char2int 1)
   (locate 2))
  ;; Formals
  (key a b)
  ;; Identities
  (('access1 (car (cons a b)) a)
   ('access2 (cdr (cons a b)) b)))

(declare-funcs secd-helpers
  () ; no paramerters
  ;; Function signatures
  ([dcd ("lt") "secd-state"]
   [test-inst? ("secd-state" "lt") "lt"]
   [true? ("lt") "boolean"])
  ;; Formals
  ([a "lt"] [b "lt"] [k "lt"] [tst "boolean"])
  ;; Identities
  (['if-def       "(if k a b)"      "(sel (true? k) a b)"]
   ['test?-def    "(test? a b)"     "(test-inst? (dcd a) b)"]
   ['test-inst?-def-atom "(test-inst? atom-t b)" "(atom? b)"]
   ['test-inst?-def-num  "(test-inst? num-t b)"  "(num? b)"]
   ['test-inst?-def-sym  "(test-inst? sym-t b)"  "(sym? b)"]
   ['test-inst?-def-pair "(test-inst? pair-t b)" "(pair? b)"]))
```

Figure 23: SECD specification data-types

```
(setcar!*
  (setcdr!*
    (setcar!*
      (setcdr!*
        (setcar!*
          (setcdr!*
            (setcar!*
              (setcdr!* mem (alloc* mem) (cdr* mem (car* mem s*)))
              (alloc* mem)
              (car* mem (cdr* mem s*)))
            (alloc*
              (setcar!*
                (setcdr!* mem (alloc* mem) (cdr* mem (car* mem s*)))
                (alloc* mem)
                (car* mem (cdr* mem s*))))
            d*)
          (alloc*
            (setcar!*
              (setcdr!* mem (alloc* mem) (cdr* mem (car* mem s*)))
              (alloc* mem)
              (car* mem (cdr* mem s*))))
          (cdr* (setcar!*
                  (setcdr!* mem (alloc* mem) (cdr* mem (car* mem s*)))
                  (alloc* mem)
                  (car* mem (cdr* mem s*)))
                c*))
        (alloc*
          (setcar!*
            (setcdr!*
              (setcar!*
                (setcdr!* mem (alloc* mem) (cdr* mem (car* mem s*)))
                (alloc* mem)
                (car* mem (cdr* mem s*)))
              (alloc* ...
```

Figure 24: The first 15% of the memory term from branch `ap-t` had refinement been applied prior to any serialization

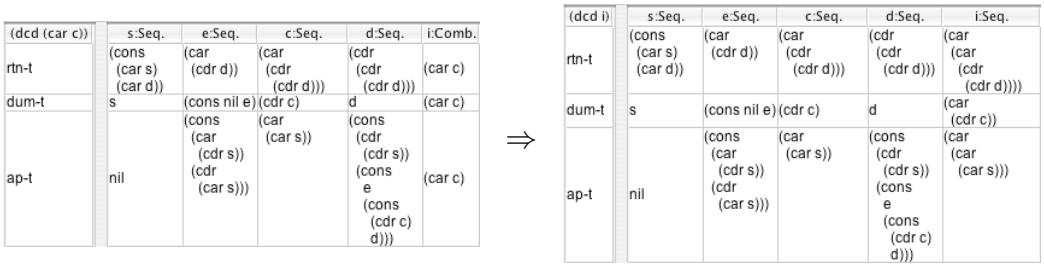


Figure 25: The first table fragment introduces a combinational signal `i` that if uniformly equal to `(car c)`. Intuitively, deriving the second table entails combinational identification to replace the guard `(dcd (car c))` with `(dcd i)` followed by converting `i` to a sequential signal. However well-formedness disallows combinational signals from guard expressions, thus the derivation follows the more arcane sequence described in Section 9.2.4.

9.2.4 Introducing fetch

WS places, or *fetches*, each instruction into a register `i`. HS guards its actions with `(dcd (car c))`. The goal is to guard it with `(dcd i)` where `i` is a register. To do this: collapse the guard, so that there is only one action row. The collapse replaces table rows with a single $sel(dcd(carc))...$) whose branches are the specification table's row terms. Then introduce a combinational signal `i` that equals `(car c)`. Use combinational substitution to replace the selector key `(dcd (car c)` by `(dcd i)`. Transform `i` into a sequential signal, and finally expand back to a table over the selection key `(dcd i)`. The update actions for `i` mirror those for `c`, except that they additionally `car`-access the expressions. After serialization, this additional access becomes the fetch step at the tail of every case (Figure 25).

9.2.5 Expanding functions

HS uses the functions `atom?`, `num?`, `sym?`, and `pair?`. The target expression consolidates these predicates with the parameterized function `test?`, which uses the current instruction (from register `i`) to select the operation. Using the identities from the `secd-helpers` declaration (Figure 23) and the guard identity, the derivation replaces `atom?` with `test?` (case `atom-t`, signal `s`):

```
(atom? (car s)) = (test-inst? atom-t (car s))
                = (test-inst? (dcd i) (car s))
                = (test? i (car s))
```

Similar reasoning allows replacement of the other three predicates by `test?` in cases `num-t`, `sym-t`, and `pair-t`. The pre-serialization strategy also converts `if` to its `sel` counterpart (case `sel-t`, signal `c`):

```
(if (car s)                (sel (true? (car s))
  (car (cdr c))        =     (car (cdr c))
  (car (cdr (cdr c))))       (car (cdr (cdr c))))

                           (car
                       =     (sel (true? (car s))
                               (cdr c)
                               (cdr (cdr c))))
```

9.2.6 Initial serialization and scheduling

The next task schedules `cons`, `car` and `cdr`, to apply at most one instance per step. A second sequential signal, `j`, combined with `i` hold most intermediate results. Scheduling matches WS as closely as possible. Since WS operates on the heap representation, `cons` applications could not be split up in the same manner—as it is a sequence of `alloc`, `setcar!` and `setcdr!`.

```
(exec
  (lambda (s e c d mem i j do_gc donesecd sio)
    (case i
      (RTN-t  (rtn1 ? ? ? d mem (car* mem s) ? (bit nil)(bit nil) sio))
      ...
      )))
(rtn1 (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn2 ? ? ? d mem i (alloc* mem)  (bit nil)(bit nil) sio)))
(rtn2 (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn2.4 j ? ? d mem i ? (bit nil)(bit nil) sio)))
(rtn2.4 (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn2.5 s ? ? d mem i (car* mem d)  (bit nil)(bit nil) sio)))
(rtn2.5 (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn2.8 s ? ? d (setcar!* mem s i) ? j (bit nil)(bit nil) sio)))
(rtn2.8 (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn3 s ? ? d (setcdr!* mem s j) ? ? (bit nil)(bit nil) sio)))
(rtn3 (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn3.5 s ? ? d mem ? (cdr* mem d)  (bit nil)(bit nil) sio)))
(rtn3.5 (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn4 s ? ? d mem (car* mem j) j  (bit nil)(bit nil) sio)))
(rtn4 (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn4.5 s ? ? d mem i (cdr* mem j)  (bit nil)(bit nil) sio)))
(rtn4.5 (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn5 s i ? d mem ? j (bit nil)(bit nil) sio)))
(rtn5 (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn6 s e ? d mem (car* mem j) j  (bit nil)(bit nil) sio)))
(rtn6 (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn6.5 s e i d mem ? j (bit nil)(bit nil) sio)))
(rtn6.5 (lambda (s e c d mem i j do_gc donesecd sio)
    (rtn7 s e c d mem (cdr* mem j) ?  (bit nil)(bit nil) sio)))
(rtn7 (lambda (s e c d mem i j do_gc donesecd sio)
    (fetch s e c i mem ? ? (bit nil)(bit nil) sio)))
(fetch (lambda (s e c d mem i j do_gc donesecd sio)
  (if (need_2_gc)
   (init_gc1 s e c d mem (alloc* mem) ? (bit nil)(bit nil) sio)
   (exec s e c d mem (car* mem c) ? (bit nil)(bit nil) sio))))
```

Figure 26: The WS schedule of `rtn-t` splits the execution of a `cons` (`alloc*`, `setcar!*`, `setcdr!*` corresponding to steps `rtn1`, `rtn2.5` and `rtn2.8`) over storage in `s` (`rtn2`) computation of `(car* mem d)` (`rtn2.4`).

ser	(dcd i)	s:Seq.	e:Seq.	c:Seq.	d:Seq.	i:Seq.	j:Seq.
0	rtn-t	#	#	#	d	(car s)	#
1	rtn-t	#	#	#	d	i	(car d)
2	rtn-t	(cons i j)	#	#	d	#	#
3	rtn-t	s	#	#	d	#	(cdr d)
4	rtn-t	s	#	#	d	(car j)	j
5	rtn-t	s	#	#	d	i	(cdr j)
6	rtn-t	s	i	#	d	#	j
7	rtn-t	s	e	#	d	(car j)	j
8	rtn-t	s	e	i	d	#	j
9	rtn-t	s	e	c	d	(cdr j)	#
10	rtn-t	s	e	c	i	#	#
11	rtn-t	s	e	c	d	(car c)	#

Figure 27: This approximates the WS schedule of `rtn-t` (Figure 26); the level of granularity requires atomic execution for`cons`.

For example, WS's schedule of `rtn-t` (Figure 26) first allocates heap space in state `rtn1`, then moves the referencing address to register `s` and computes `(car* mem d)` in `j`, prior to the initialization of the heap cell. Interleaving the execution of `cons` applications with address transfers and `(car* mem d)` is not possible in the `lt` data-type. The approximation of this schedule in `lt` (Figure 27) computes `(car d)` in `j` prior to `(cons i j)` directly in register `s`. After translating to heap representation, reserialization obtains WS's schedule. Similar judgement calls are needed for other `cons` applications.

WS's schedule uses a microcode style compression of same-control suffixes. For instance, the schedules of `car-t` and `cdr-t` are identical except for the second step where `car` returns one element of the pair, while `cdr` returns the other. Rather than duplicate the shared code, `cdr1` transitions to the state `car2`. Similarly the end of the `car` schedule shares a control suffix with other schedules as indicated by the transfer to state `next1` (Figure 28). It is not possible to express this kind of suffix sharing with the serialization tables, thus the Starfish derivation enumerates each schedule in

```
(exec
  (lambda (s e c d mem i j do_gc donesecd sio)
    (case i
     (CAR-t  (car1 s e c d mem (car* mem s) ? (bit nil)(bit nil) sio))
     (CDR-t  (cdr1 s e c d mem (car* mem s) ? (bit nil)(bit nil) sio))
      ...
      )))
(car1 (lambda (s e c d mem i j do_gc donesecd sio)
  (car2 s e c d mem (car* mem i) ? (bit nil)(bit nil) sio)))
(car2 (lambda (s e c d mem i j do_gc donesecd sio)
  (car3 s e c d mem i (alloc* mem)(bit nil)(bit nil) sio)))
(car3 (lambda (s e c d mem i j do_gc donesecd sio)
  (car4 s e c d (setcar!* mem j i) ? j (bit nil)(bit nil) sio)))
(car4 (lambda (s e c d mem i j do_gc donesecd sio)
  (car5 ? e c d mem (cdr* mem s) j (bit nil)(bit nil) sio)))
(car5 (lambda (s e c d mem i j do_gc donesecd sio)
  (car6 ? e c d (setcdr!* mem j i) ? j (bit nil)(bit nil) sio)))
(car6 (lambda (s e c d mem i j do_gc donesecd sio)
  (next1 j e c d mem ? ? (bit nil)(bit nil) sio)))
(cdr1 (lambda (s e c d mem i j do_gc donesecd sio)
  (car2 s e c d mem (cdr* mem i) ? (bit nil)(bit nil) sio)))
(next1 (lambda (s e c d mem i j do_gc donesecd sio)
  (next2 s e ? d mem (cdr* mem c) ? (bit nil)(bit nil) sio)))
(next2 (lambda (s e c d mem i j do_gc donesecd sio)
  (fetch s e i d mem ? ? (bit nil)(bit nil) sio)))
(fetch (lambda (s e c d mem i j do_gc donesecd sio)
  (if (need_2_gc)
   (init_gc1 s e c d mem (alloc* mem) ? (bit nil)(bit nil) sio)
   (exec s e c d mem (car* mem c) ? (bit nil)(bit nil) sio))))
```

ser (dcd i)		s*:Seq.	e*:Seq.	c*:Seq.	d*:Seq.	i*:Seq.	j*:Seq.	mem:Seq.
0	car-t	s*	e*	c*	d*	(car* mem s*)	#	mem
1	car-t	s*	e*	c*	d*	(car* mem i*)	#	mem
2	car-t	s*	e*	c*	d*	i*	(alloc* mem)	mem
3	car-t	s*	e*	c*	d*	#	j*	(setcar!* mem j* i*)
4	car-t	#	e*	c*	d*	(cdr* mem s*)	j*	mem
5	car-t	#	e*	c*	d*	#	j*	(setcdr!* mem j* i*)
6	car-t	j*	e*	c*	d*	#	#	mem
7	car-t	s*	e*	#	d*	(cdr* mem c*)	#	mem
8	car-t	s*	e*	i*	d*	#	#	mem
9	car-t	s*	e*	c*	d*	(car* mem c*)	#	mem

ser (dcd i)		s*:Seq.	e*:Seq.	c*:Seq.	d*:Seq.	i*:Seq.	j*:Seq.	mem:Seq.
0	cdr-t	s*	e*	c*	d*	(car* mem s*)	#	mem
1	cdr-t	s*	e*	c*	d*	(cdr* mem i*)	#	mem
2	cdr-t	s*	e*	c*	d*	i*	(alloc* mem)	mem
3	cdr-t	s*	e*	c*	d*	#	j*	(setcar!* mem j* i*)
4	cdr-t	#	e*	c*	d*	(cdr* mem s*)	j*	mem
5	cdr-t	#	e*	c*	d*	#	j*	(setcdr!* mem j* i*)
6	cdr-t	j*	e*	c*	d*	#	#	mem
7	cdr-t	s*	e*	#	d*	(cdr* mem c*)	#	mem
8	cdr-t	s*	e*	i*	d*	#	#	mem
9	cdr-t	s*	e*	c*	d*	(car* mem c*)	#	mem

Figure 28: WS reuses common control suffixes to compress its expressions. The `cdr1` state transfers control to `car2`. Many of the serial control flows end with a transfer to `next1` or `fetch`. However, Starfish's serialization must explicitly enumerate all common suffixes as shown by the final, post data-refinement schedules for `car-t` and `cdr-t`.

```
...
(ld3 (lambda (s e c d mem i j do_gc donesecd sio)
    (if (zero?* i)
      (ld4 s e c d mem ? (car* mem j)(bit nil)(bit nil) sio)
      (ld3.5 s e c d mem i (cdr* mem j)(bit nil)(bit nil) sio))))
(ld3.5 (lambda(s e c d mem i j do_gc donesecd sio)
    (ld3 s e c d mem (sub1* i) j (bit nil)(bit nil) sio)))
(ld4 (lambda (s e c d mem i j do_gc donesecd sio)
    (ld5 s e c d mem (cdr* mem c) j (bit nil)(bit nil) sio)))
(ld5 (lambda (s e c d mem i j do_gc donesecd sio)
    (ld6 s e c d mem (car* mem i) j (bit nil)(bit nil) sio)))
(ld6 (lambda (s e c d mem i j do_gc donesecd sio)
    (ld7 s e c d mem (cdr* mem i) j (bit nil)(bit nil) sio)))
(ld7 (lambda (s e c d mem i j do_gc donesecd sio)
    (if (zero?* i)
      (ld8 s e c d mem (car* mem j) ? (bit nil)(bit nil) sio)
      (ld7.5 s e c d mem i (cdr* mem j)(bit nil)(bit nil) sio))))
(ld7.5 (lambda (s e c d mem i j do_gc donesecd sio)
    (ld7 s e c d mem (sub1* i) j (bit nil)(bit nil) sio)))
(ld8 (lambda (s e c d mem i j do_gc donesecd sio)
    (ld10 s e c d mem i (alloc* mem)(bit nil)(bit nil) sio)))
...
```

Figure 29: WS embeds a looping control flow that implements `(locate* mem i j)` with control states `ld3` through `ld7.5`.

toto.

WS's schedule of `ld-t` embeds a control loop for `locate`, a combinator which calls the iterative recursive function `index`. This sort of iterative looping is not possible with Starfish's serialization facility. Instead, the derivation uses `locate` as a primitive to be factored out of the primary controller.

WS routinely places don't-care markers in registers that are no longer live. I noted a missed opportunity in the serialization of `ap-t`. After step `9` of the initial schedule, the contents of `d` no longer effect the result (Figure 30). Starfish's initial serialization marks these don't-care's. The serialization table confirms the correctness

ser	(dcd i)	s:Seq.	e:Seq.	c:Seq.	d:Seq.	i:Seq.	j:Seq.
0	ap-t	s	e	c	d	#	(car s)
1	ap-t	s	e	c	d	#	(cdr j)
2	ap-t	s	e	c	d	(cdr s)	j
3	ap-t	s	e	c	d	(car i)	j
4	ap-t	s	e	c	d	(cons i j)	j
5	ap-t	s	#	c	d	i	e
6	ap-t	s	i	c	d	#	j
7	ap-t	s	e	#	d	(cdr c)	j
8	ap-t	s	e	j	d	i	#
9	ap-t	s	e	c	#	#	(cons i d)
10	ap-t	s	e	#	#	(cons c j)	#
11	ap-t	s	e	#	#	i	(cdr s)
12	ap-t	s	e	#	#	i	(cdr j)
13	ap-t	s	e	#	#	#	(cons j i)
14	ap-t	s	e	#	j	#	#
15	ap-t	#	e	#	d	(car s)	#
16	ap-t	#	e	#	d	(car i)	#
17	ap-t	#	e	i	d	#	#
18	ap-t	nil	e	c	d	#	#
19	ap-t	s	e	c	d	(car c)	#

Figure 30: The value of `d` is no longer needed after step `9`; it is replaced with a `#`. WS exploits similar optimizations, although this particular one was overlooked.

of this simplification upon commitment to the behavior table.

9.2.7 Data refinement

HS operates on abstract pairs over atoms, defined by `lt`. This stage of the derivation translates the system into WS's native data-type, replacing the abstract pairs (`lt`) with cell references (`lc`) to a heap (`lmem{lc}`). The `lc` declaration only declares function signatures and constants with no identities. Intuitively the `lc`, for *LISP cell* is a tagged integer, where the tag specifies *empty-list, character, integer, symbol* or *pair*. The first three classifications are atoms; their numerical portion does not reference the heap. The numerical field of *pair* or *symbol* tagged cell is a reference to the heap. Symbols for this implementation are lists of characters whose head element

```
(define-param-alg lmem
  (cell)
  ()
  ([car* (lmem cell) cell]
   [cdr* (lmem cell) cell]
   [alloc* (lmem) cell]
   [setcar!* (lmem cell cell) lmem]
   [setcdr!* (lmem cell cell) lmem])
  ([m lmem] [i cell] [j cell] [k cell] [l cell])
  (['car*-setcdr!* (car* (setcdr!* m j k) i) (car* m i)]
   ['cdr*-setcar!* (cdr* (setcar!* m j k) i) (cdr* m i)]
   ['setcar!*-setcdr!*-comm
     (setcar!* (setcdr!* m i j) k l)
     (setcdr!* (setcar!* m k l) i j)]))
(define-term-alg lc
  (nil* err* fls* tru*)
  ((nil?* 1) (atom?* 1) (num?* 1) (pair?* 1) (sym?* 1)
   (test?* 2)
   (eq?* 2) (<=?* 2) (plus* 2) (minus* 2)
   (sym-list* 1) (char2int* 1)) () ())
```

Figure 31: Signature declarations for a heap memory.

carries a *symbol*-tag. A further refinement could make this mapping explicit, however such precision is unnecessary for the derivation's target.

The heap structure, `lmem` for *LISP memory*, stores the cells in pairs. A pair or symbol tagged cell references one of these pairs with its numerical field. The accessors `car*` and `cdr*` (`lmem{lc}` $\times$ `lc` $\rightarrow$ `lc`) return the first and second cells respectively of the pair referenced by their cell input. The `alloc*` function (`lmem{lc}` $\rightarrow$ `lc`) returns a pair-tagged cell reference to an uninitialized pair in the heap, while the functions `setcar!*` and `setcdr!*` (`lmem{lc}` $\times$ `lc` $\times$ `lc` $\rightarrow$ `lmem{lc}`) replace their respective cells (as referenced by their middle argument) with the cell in their trailing argument. The `lmem` declaration contains three identities that show how effects and

```
(declare-funcs heap-helpers
  ()
  ([true?* ("lc") "boolean"]
   [lc-eq? ("lc" "lc") "boolean"]
   [locate* ("lmem{lc}" "lc" "lc") "lc"]
   [list-sym* ("lmem{lc}" "lc") "lc"])
  ([m "lmem{lc}"] [i "lc"] [j "lc"] [k "lc"])
  (['setcdr!*-inits-car-to-nil
     "(setcdr!* m (alloc* m) i)"
     "(setcar!* (setcdr!* m (alloc* m) i) (alloc* m) nil*)"]
   ['setcar!*-inits-cdr-to-nil
     "(setcar!* m (alloc* m) i)"
     "(setcdr!* (setcar!* m (alloc* m) i) (alloc* m) nil*)"]
   ['setcdr!*-effects-mem
     "(cdr* (setcdr!* m i j) k)"
     "(sel (lc-eq? i k) j (cdr* m k))"]
   ['setcar!*-effects-mem
     "(car* (setcar!* m i j) k)"
     "(sel (lc-eq? i k) j (car* m k))"]
   ['lc-eq?-comm "(lc-eq? i j)" "(lc-eq? j i)"]
   ['lc-eq?-true "(lc-eq? i i)" "true"]
   ['lc-eq?-false-1 "(lc-eq? (alloc* m) (car* m i))" "false"]
   ['lc-eq?-false-2 "(lc-eq? (alloc* m) (cdr* m i))" "false"]))
```

Figure 32: These remaining functions and identities round-out the necessary signature for the heap.

accessors commute with each other. These identities are critical to justifying the WS memory access schedule. The first two indicate that an arbitrary `setcdr!*` change to memory does not alter any subsequent `car*` access, and similarly `setcar!*`'s changes are invisible to future applications of `cdr*`. The third identity shows that arbitrary `setcar!*` and `setcdr!*` updates to the heap are independent and may commute.

The next declarative block, `heap-helpers`, defines the predicate signature `lc-eq?` and heap versions of `true?`, `locate` and `list-sym`. `lc-eq?` intuitively returns the boolean true when it's arguments are the same and false otherwise. `locate` and

`list-sym` need to traverse paths within their potentially nested pair arguments, thus their heap implementations require an additional argument for the heap itself. Four identities constrain `lc-eq?`'s behavior to return `true` on equal elements (reflexivity), `false` for comparision of newly allocated cells with existing cells (i.e., those which may be accessed in the memory prior to allocation), and that its arguments commute (symmetry). The `setcXr!*-effects` identities show when heap updates change access results of a given reference. These identities validate much of the retiming in Section 9.2.8. Two more identities in this block prescribe an unintialized field in a newly allocated pair to be `nil` *after* initialization of the other field. These identities are critical in post-refinement scheduling.

The data refinement uses the state-threading schema presented in Section 8.3 to map `lt` to `lmemlc`. The core identities to this homomorphism relate `cons`, `car` and `cdr` to their implementations in `lmem{lc}`. The following diagrams show how `lt`'s functions are embedded in the refinement type. They commute with respect to abstract equivalence; i.e.,

$$[r_0\ m_0] \equiv [r_1\ m_1] \Leftrightarrow \alpha(r_0, m_0) = \alpha(r_1, m_1) \tag{162}$$

The following bindings hold for the diagrams below:

```
[a* ma]  = (pr=>heap a m)
[b* mab] = (pr=>heap b ma)
```

```
(declare-refinement pr => cell
  () "lt" "lc" ([m "lmem{lc}" "#:lmem{lc}"])
  ;; Formals
  ([a "lt"] [b "lt"] [i "lc"] [j "lc"] [tst "boolean"])
  ;; Image bindings
  ([a*  "(pr=>cell   a m)"]
   [ma  "(pr=>cell.m a m)"]
   [b*  "(pr=>cell   b (pr=>cell.m a m))"]
   [mab "(pr=>cell.m b (pr=>cell.m a m))"])
  ;; Refinement for functions that produce lt's
  ( ["#:lt" "#:lc" "m"]
    ["(cons a b)" "(alloc* mab)"
                  "(setcar!* (setcdr!* mab (alloc* mab) b*)
                    (alloc* mab) a*)"]
    ["(car a)" "(car* ma a*)" "ma"]
    ["(cdr a)" "(cdr* ma a*)" "ma"]
    ["(nil? a)" "(nil?* a*)" "ma"]
    ["(atom? a)" "(atom?* a*)" "ma"]
    ["(num? a)" "(num?* a*)" "ma"]
    ["(pair? a)" "(pair?* a*)" "ma"]
    ["(sym? a)" "(sym?* a*)" "ma"]
    ["(test? a b)" "(test?* a* b*)" "mab"]
    ["(eq? a b)" "(eq?* a* b*)" "mab"]
    ["(<=? a b)" "(<=?* a* b*)" "mab"]
    ["(plus a b)" "(plus* a* b*)" "mab"]
    ["(minus a b)" "(minus* a* b*)" "mab"]
    ["(char2int a)" "(char2int* a*)" "ma"]
    ["(sym-list a)" "(sym-list* a*)" "ma"]
    ["(list-sym a)" "(list-sym* ma a*)" "ma"]
    ["(locate a b)" "(locate* mab a* b*)" "mab"]
    ["nil" "nil*" "m"]
    ["err" "err*" "m"]
    ["fls" "fls*" "m"]
    ["tru" "tru*" "m"])
  ;; For functions that consume lt's, but do not produce lt's?
  (['pr=>cell:true? "(true? a)" "(true?* a*)"]))
```

Figure 33: Refinement declaration from pairs to heap representation. Identities are expressed as the top and bottom terms of the commuting diagram as in Section 9.2.7.

$$\begin{array}{ccc} \texttt{lt} \times \texttt{lmem}\{\texttt{lc}\} & \xrightarrow{\texttt{[(cons a b) m]}} & \texttt{lt} \times \texttt{lmem}\{\texttt{lc}\} \\ {\scriptstyle \texttt{pr=>heap}}\downarrow & & \downarrow{\scriptstyle \texttt{pr=>heap}} \\ \texttt{lc} \times \texttt{lmem}\{\texttt{lc}\} & \xrightarrow[\substack{\texttt{[(alloc mab)}\\ \texttt{(setcar!*}\\ \texttt{(setcdr!* m* (alloc mab) a*)}\\ \texttt{(alloc mab)}\\ \texttt{b*)]}}]{} & \texttt{lc} \times \texttt{lmem}\{\texttt{lc}\} \end{array} \tag{163}$$

$$\begin{array}{ccc} \texttt{lt} \times \texttt{lmem}\{\texttt{lc}\} & \xrightarrow{\texttt{[(car a) m]}} & \texttt{lt} \times \texttt{lmem}\{\texttt{lc}\} \\ {\scriptstyle \texttt{pr=>heap}}\downarrow & & \downarrow{\scriptstyle \texttt{pr=>heap}} \\ \texttt{lc} \times \texttt{lmem}\{\texttt{lc}\} & \xrightarrow[\texttt{[(car* ma a*) ma]}]{} & \texttt{lc} \times \texttt{lmem}\{\texttt{lc}\} \end{array} \tag{164}$$

$$\begin{array}{ccc} \texttt{lt} \times \texttt{lmem}\{\texttt{lc}\} & \xrightarrow{\texttt{[(cdr a) m]}} & \texttt{lt} \times \texttt{lmem}\{\texttt{lc}\} \\ {\scriptstyle \texttt{pr=>heap}}\downarrow & & \downarrow{\scriptstyle \texttt{pr=>heap}} \\ \texttt{lc} \times \texttt{lmem}\{\texttt{lc}\} & \xrightarrow[\texttt{[(cdr* ma a*) ma]}]{} & \texttt{lc} \times \texttt{lmem}\{\texttt{lc}\} \end{array} \tag{165}$$

Similarly, he functions `list-sym` and `locate` map to their counterparts. They do not change the heap, but need it for evaluation.

$$\begin{array}{ccc} \texttt{lt} \times \texttt{lmem}\{\texttt{lc}\} & \xrightarrow{\texttt{[(list-sym a) m]}} & \texttt{lt} \times \texttt{lmem}\{\texttt{lc}\} \\ {\scriptstyle \texttt{pr=>heap}}\downarrow & & \downarrow{\scriptstyle \texttt{pr=>heap}} \\ \texttt{lc} \times \texttt{lmem}\{\texttt{lc}\} & \xrightarrow[\texttt{[(list-sym* ma a*) ma]}]{} & \texttt{lc} \times \texttt{lmem}\{\texttt{lc}\} \end{array} \tag{166}$$

$$\begin{array}{ccc} \texttt{lt} \times \texttt{lmem}\{\texttt{lc}\} & \xrightarrow{\texttt{[(locate a b) m]}} & \texttt{lt} \times \texttt{lmem}\{\texttt{lc}\} \\ {\scriptstyle \texttt{pr=>heap}}\downarrow & & \downarrow{\scriptstyle \texttt{pr=>heap}} \\ \texttt{lc} \times \texttt{lmem}\{\texttt{lc}\} & \xrightarrow[\texttt{[(locate* mab a* b*) mab]}]{} & \texttt{lc} \times \texttt{lmem}\{\texttt{lc}\} \end{array} \tag{167}$$

The remaining functions and constants in `lt` correspond to their starred counterparts in `lc` with no effect on the heap and no need for heap access. The schema for one-

and two-argument function identities follows:

$$\begin{array}{ccc} \texttt{lt} \times \texttt{lmem}\{\texttt{lc}\} & \xrightarrow{\texttt{[(f a) m]}} & \texttt{lt} \times \texttt{lmem}\{\texttt{lc}\} \\ {\scriptstyle\texttt{pr=>heap}}\downarrow & & \downarrow{\scriptstyle\texttt{pr=>heap}} \\ \texttt{lc} \times \texttt{lmem}\{\texttt{lc}\} & \xrightarrow[\texttt{[(f a*) ma]}]{} & \texttt{lc} \times \texttt{lmem}\{\texttt{lc}\} \end{array} \tag{168}$$

$$\begin{array}{ccc} \texttt{lt} \times \texttt{lmem}\{\texttt{lc}\} & \xrightarrow{\texttt{[(h a b) m]}} & \texttt{lt} \times \texttt{lmem}\{\texttt{lc}\} \\ {\scriptstyle\texttt{pr=>heap}}\downarrow & & \downarrow{\scriptstyle\texttt{pr=>heap}} \\ \texttt{lc} \times \texttt{lmem}\{\texttt{lc}\} & \xrightarrow[\texttt{[(h a* b*) mab]}]{} & \texttt{lc} \times \texttt{lmem}\{\texttt{lc}\} \end{array} \tag{169}$$

Starfish's data refinement facility also allows the user to specify automatic rewrites of functions that do not produce `lt`s yet accept `lt` parameters. The one translation in this category is `true?`

$$\begin{array}{ccc} \texttt{lt} \times \texttt{lmem}\{\texttt{lc}\} & \xrightarrow{\texttt{[(true? a) m]}} & \texttt{boolean} \\ {\scriptstyle\texttt{pr=>heap}}\downarrow & & \| \\ \texttt{lc} \times \texttt{lmem}\{\texttt{lc}\} & \xrightarrow[\texttt{[(true?* a*) ma]}]{} & \texttt{boolean} \end{array} \tag{170}$$

Starfish automatically rewrites terms using these identities. Since memory accesses are already serialized, no two registers try to update `mem` in the same action. When this is not the case, Starfish follows the signal order specified in the transformation command for updating the state.

9.2.8 Re-serialization and re-scheduling

Data translation from abstract pairs to a referenced heap exposes the memory accesses resulting from application of `cons`. Reserialization separates the `alloc*`, `setcar!*`, and `setcdr!*` steps in `cons`'s implementation.

Initial schedule of `rtn-t` after conversion from pairs to heap

ser	(dcd i)	s*:Seq.	e*:Seq.	c*:Seq.	d*:Seq.	i*:Seq.	j*:Seq.	mem:Seq.
0	rtn-t	#	#	#	d*	(car* mem s*)	#	mem
1	rtn-t	#	#	#	d*	i*	(car* mem d*)	mem
2	rtn-t	(alloc* mem)	#	#	d*	#	#	(setcar!* (setcdr!* mem (alloc* mem) j*) (alloc* mem) i*)
3	rtn-t	s*	#	#	d*	#	(cdr* mem d*)	mem
4	rtn-t	s*	#	#	d*	(car* mem j*)	j*	mem
5	rtn-t	s*	#	#	d*	i*	(cdr* mem j*)	mem
6	rtn-t	s*	i*	#	d*	#	j*	mem
7	rtn-t	s*	e*	#	d*	(car* mem j*)	j*	mem
8	rtn-t	s*	e*	i*	d*	#	j*	mem
9	rtn-t	s*	e*	c*	d*	(cdr* mem j*)	#	mem
10	rtn-t	s*	e*	c*	i*	#	#	mem
11	rtn-t	s*	e*	c*	d*	(car* mem c*)	#	mem

Insertion point of limited reserialization: Note that the serial for lines `1` and `2` have been symbolically combined into a single row of subject terms.

ser	(dcd i)	s*:Seq.	e*:Seq.	c*:Seq.	d*:Seq.	i*:Seq.	j*:Seq.	mem:Seq.
0	rtn-t	#	#	#	d*	(car* mem s*)	#	mem
insert	rtn-t	(alloc* mem)	#	#	d*	#	#	(setcar!* (setcdr!* mem (alloc* mem) (car* mem d*)) (alloc* mem) i*)
3	rtn-t	s*	#	#	d*	#	(cdr* mem d*)	mem
4	rtn-t	s*	#	#	d*	(car* mem j*)	j*	mem
5	rtn-t	s*	#	#	d*	i*	(cdr* mem j*)	mem
6	rtn-t	s*	i*	#	d*	#	j*	mem
7	rtn-t	s*	e*	#	d*	(car* mem j*)	j*	mem
8	rtn-t	s*	e*	i*	d*	#	j*	mem
9	rtn-t	s*	e*	c*	d*	(cdr* mem j*)	#	mem
10	rtn-t	s*	e*	c*	i*	#	#	mem
11	rtn-t	s*	e*	c*	d*	(car* mem c*)	#	mem

Serialization table: The lhs shows the proposed schedule, while the rhs shows its serial symbolic evaluation. The last row shows the subject terms which the serialization's evaluation must achieve. The last step (not shown) commutes the order of `setcar!*` and `setcdr!*` in the fifth row of the rhs.

s*	e*	c*	d*	i*	j*	mem
#	#	#	d*	i*	(alloc* mem)	mem
j*	#	#	d*	i*	#	mem
s*	#	#	d*	i*	(car* mem d*)	mem
s*	#	#	d*	#	j*	(setcar!* mem s* i*)
s*	#	#	d*	#	#	(setcdr!* mem s* j*)
#	#	#	#	#	#	#

s*	e*	c*	d*	i*	j*	mem
#	#	#	d*	i*	(alloc* mem)	mem
(alloc* mem)	#	#	d*	i*	#	mem
(alloc* mem)	#	#	d*	i*	(car* mem d*)	mem
(alloc* mem)	#	#	d*	#	(car* mem d*)	(setcar!* mem (alloc* mem) i*)
(alloc* mem)	#	#	d*	#	#	(setcdr!* (setcar!* mem (alloc* mem) i*) (alloc* mem) (car* mem d*))
(alloc* mem)	#	#	d*	#	#	(setcar!* (setcdr!* mem (alloc* mem) (car* mem d*)) (alloc* mem) i*)

Result of commiting reserialization

ser	(dcd i)	s*:Seq.	e*:Seq.	c*:Seq.	d*:Seq.	i*:Seq.	j*:Seq.	mem:Seq.
0	rtn-t	#	#	#	d*	(car* mem s*)	#	mem
1	rtn-t	#	#	#	d*	i*	(alloc* mem)	mem
2	rtn-t	j*	#	#	d*	i*	#	mem
3	rtn-t	s*	#	#	d*	i*	(car* mem d*)	mem
4	rtn-t	s*	#	#	d*	#	j*	(setcar!* mem s* i*)
5	rtn-t	s*	#	#	d*	#	#	(setcdr!* mem s* j*)
6	rtn-t	s*	#	#	d*	#	(cdr* mem d*)	mem
7	rtn-t	s*	#	#	d*	(car* mem j*)	j*	mem
8	rtn-t	s*	#	#	d*	i*	(cdr* mem j*)	mem
9	rtn-t	s*	i*	#	d*	#	j*	mem
10	rtn-t	s*	e*	#	d*	(car* mem j*)	j*	mem
11	rtn-t	s*	e*	i*	d*	#	j*	mem
12	rtn-t	s*	e*	c*	d*	(cdr* mem j*)	#	mem
13	rtn-t	s*	e*	c*	i*	#	#	mem
14	rtn-t	s*	e*	c*	d*	(car* mem c*)	#	mem

Figure 34: Reserializing steps `1` and `2` of `rtn-t`: The refinement of pairs to heaps replaces `cons` with a sequence of `alloc*`, `setcar!*` and `setcdr!*`. Starfish only commits a serialization to the behavior table when the serial evaluation matches the subject terms. Identity application is frequently necessary to satisfy this requirement (as above).

The symbolic evaluation of target schedules differ in several places from the terms at this stage of the derivation. To achieve the target schedules, we use algebraic identities in the evaluation tables (left half of the serialization tables) to justify correctness. In all but one case, the identities declared in the previous section suffice.

Reserializing the `cons` operation in `rtn-t` is representative of this process. Figure 34 shows the schedule for `rtn-t`. With the exception of steps 1 and 2, the schedule represents the derivation's target. Reserialization spans steps 1 and 2, where target terms are the symbolic evaluation of these two steps. The result of the subschedule is to place `(alloc* mem)` in `s*`, hold the value for `d*`, and initialize the new cell in `mem` with `i*` and `(car* mem d*)`. A hand-chosen rescheduling produces the third table in Figure 34, where the bottom row represents the target terms. The final term in `mem` differs from the target by the order of pair initialization. An application of identity `setcar!*-setcdr!*-comm` identity rewrites this term to match the target:

```
(setcdr!*                              =  (setcar!*
 (setcar!* mem (alloc* mem) i*)             (setcdr!* mem
  (alloc* mem)                               (alloc* mem)
  (car mem d*))                              (car mem d*))
                                             (alloc* mem) i*)
```

Once the last row matches the evaluation requirement, Starfish can insert the subschedule into the behavior table (Figure 34).

The subschedule for `dum-t` omits explicit intialization of the `car` field with `nil` (Figure 35). The identity `setcdr!*-inits-car-to-nil` justifies this shortcut. While `setcdr!*` is not necessarily responsible for initializing the `car` component to `nil`, the identity ensures that its value is `nil` after initializing the `cdr`. Burger's Scheme

Insertion point for reserialization

Full schedule proposal (lhs): the `mem` evaluation term needs to reflect a `nil` in the `car` cell.

`setcdr!*-inits-car-to-nil` fixes the `mem` evaluation term.

Insertion of subschedule

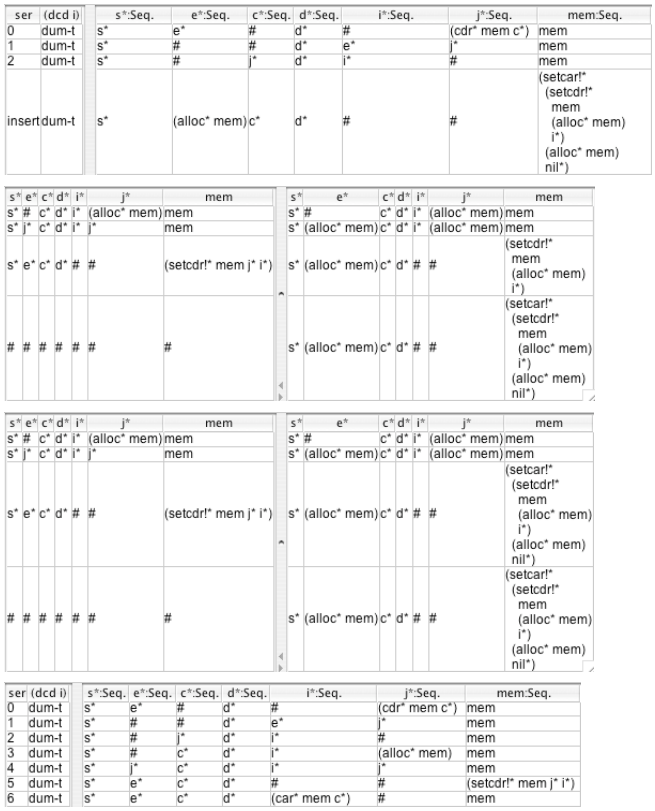


Figure 35: Reserialization of `dum-t`

machine had an independently designed process to `nil`-out or invalidate memory after garbage collection. A functional model of `alloc*` can't change the heap because it returns a reference, not a memory.

The post-translation target schedules frequently retime heap accesses to interleave with evaluation of the `alloc*-setcar!*-setcdr!*` image of `cons`. In the previous example, the target schedule retimes `(car* mem d)` from evaluation before `alloc*` to evaluation after the `alloc*`, but before the subsequent `setcar!*` and `setcdr!*`. Since `alloc*` doesn't change the `mem` register, the subsequent `cdr*` accesses the same value from `mem`. However, some of the other target schedules (e.g., `ap-t`) retime

memory accesses to after the `setcar!*` and `setcdr!*`. These memory mutations could potentially overwrite the subject of memory access. Since the heap is partitioned into `car`-cells and `cdr`-cells, pushing a `car*` access past a `setcdr!*` (or a `cdr*` access past a `setcar!*`) can not change the access result. The identities `car*-setcdr!*` and `cdr*-setcar!*` (Figure 31) express this property.

When this is insufficient justification, the identities `setcar!*-effects-mem` and `setcdr!*-effects-mem` (Figure 32) show how access to a mutated memory works relative to the original memory. The identity asserts that the scope of the mutation is limited to the cell at the address: if the access and mutation addresses are different, then the mutation does not change the access value. In some contexts, we can use identities to show that two address values are different: for instance the result of `(alloc* m)`, is always different than the result of `(car m a)` for any choice of `m` or `a` (`lc-eq?-*` in Figure 32). In other places, the inequality is the result of a control invariant rather than a data-type invariant. Such control invariants are asserted, and then externally verified. Figures 36, 37 and 38 illustrate the three different tactics we use to justify access to a mutated heap.

The remaining reserialization tasks employ the same transformation tactics. For complete details, refer to Appendix A.

9.2.9 Comparing the derivation result with the WS

Starfish's derivation (*SD*) produces much of WS from HS. The schedules for `rtn-t`, `dum-t`, `ap-t`, `sel-t`, `join-t`, `car-t`, `cdr-t`, `cons-t`, `ldc-t`, `ldf-t`, `atom-t`, `num-t`, `sym-t`, `pair-t`, `eq-t`, `leq-t`, `add-t`, `sub-t`, `exec-t`, `pop-t`, and `stop-t` exactly

Serial sub-schedule of `rtn`, steps `2` through `4`

ser	(dcd i)	s*:Seq.	e*:Seq.	c*:Seq.	d*:Seq.	i*:Seq.	j*:Seq.	mem:Seq.
2	ap-t	s*	e*	c*	d*	(cdr* mem s*)	j*	mem
3	ap-t	s*	e*	c*	d*	(car* mem i*)	j*	mem
4	ap-t	s*	e*	c*	d*	(alloc* mem)	#	(setcar!* (setcdr!* mem (alloc* mem) j*) (alloc* mem) i*)

The first three steps of the retiming proposal results in a `cdr*` access to a `setcdr!*`-mutated memory.

s*	e*	c*	d*	i*	j*	mem
s*	e*	c*	d*	(alloc* mem)	j*	mem
s*	e*	c*	d*	i*	#	(setcdr!* mem i* j*)
s*	e*	c*	d*	i*	(cdr* mem s*)	mem
#	#	#	#	#	#	#

s*	e*	c*	d*	i*	j*	mem
s*	e*	c*	d*	(alloc* mem)	j*	mem
s*	e*	c*	d*	(alloc* mem)	#	(setcdr!* mem (alloc* mem) j*)
s*	e*	c*	d*	(alloc* mem)	(cdr* (setcdr!* mem (alloc* mem) j*) s*)	(setcdr!* mem (alloc* mem) j*)
s*	e*	c*	d*	(alloc* mem)	#	(setcar!* (setcdr!* mem (alloc* mem) j*) (alloc* mem) (car* mem (cdr* mem s*)

```
(cdr*
 (setcdr!*
  mem
  (alloc* mem)
  j*)
 s*)
```
⇒
```
     (lc-eq?
(sel  (alloc* mem)
      s*)
 j*
 (cdr* mem s*))
```
⇒
```
(sel false
 j*
 (cdr* mem s*))
```
⇒
```
(cdr* mem s*)
```

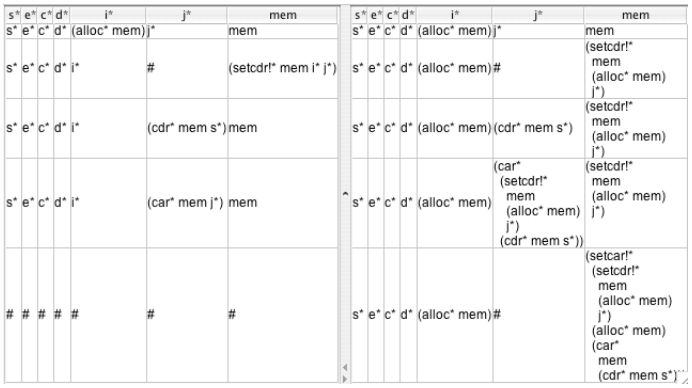
Figure 36: Retiming of `ap-t` steps `2` through `4`. The bottom row shows rewrites on `j*`'s evaluation term. The first rewrite comes from the `setcdr!*-effects-mem` identity. The next is an assertion that the two addresses are not `lc-eq?`; this becomes an external proof obligation. The final rewrite is selector evaluation. (*continued in Figure 37*)

match WS.

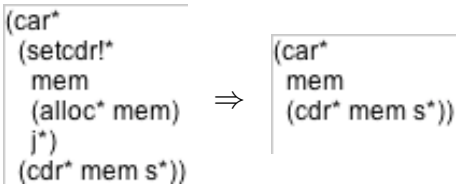
The cell-type conversion branches (`sl-t`, `ls-t`, `ci-t`) in the Starfish derivation differs from WS. WS destructively overwrites the source cell with resulting cell rather than allocating a new pair to hold the result. This is a departure from the semantics specified by HS.

The `ld-t` branch in WS contains a control loop for the behavior of `locate` in states `ld4` through `ld10`. The combination of data-refinement and serialization is one way that Starfish adds behavior to a specification—for example converting `cons` to the `alloc*-setcar!*-setcdr!*`. Starfish does not expand recursive term behavior

The retiming proposal's fourth step `car*` accesses a `setcdr!*`-mutated memory.



The `car*-setcdr!*` identity, a recognition of memory segmentation, rewrites the term as an access to the original memory.



The evaluation of the final two steps do not require any rewriting to match the subject terms. The serialization in the lhs is the new subschedule.

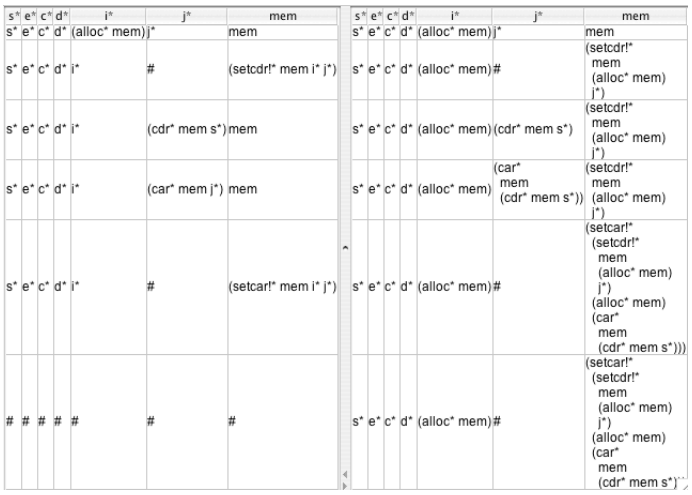
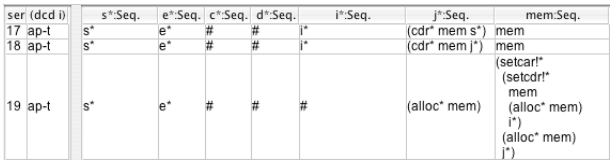


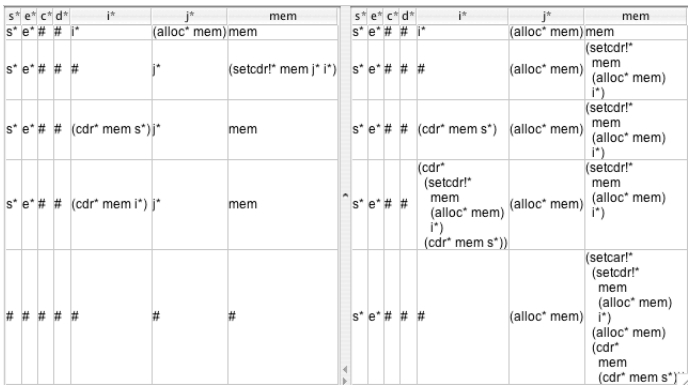
Figure 37: (*continued from Figure 36*) Retiming of `ap-t` steps `2` through `4`. Since memory is partitioned into `car` and `cdr` cells, a write to a `cdr` does not change the read of any `car`.

Serial sub-schedule of `ap`, steps `17` through `19`



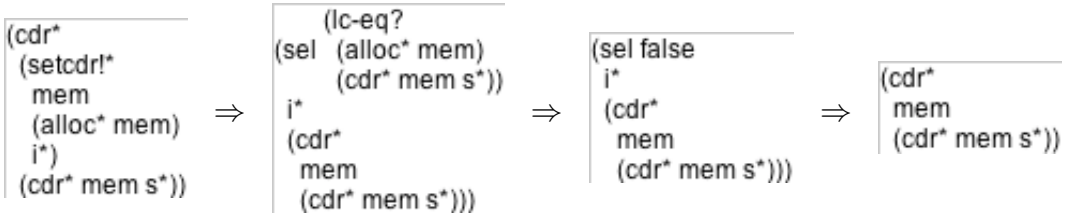
ser	(dcd i)	s*:Seq.	e*:Seq.	c*:Seq.	d*:Seq.	i*:Seq.	j*:Seq.	mem:Seq.
17	ap-t	s*	e*	#	#	i*	(cdr* mem s*)	mem
18	ap-t	s*	e*	#	#	i*	(cdr* mem j*)	mem
19	ap-t	s*	e*	#	#	#	(alloc* mem)	(setcar!* (setcdr!* mem (alloc* mem) i*) (alloc* mem) j*)

The first three steps of the retiming proposal results in `cdr*` accesses to a `setcdr!*`-mutated memory. The first access requires an external assertion, while `lc-eq?-false-1` justifies the second access (below).

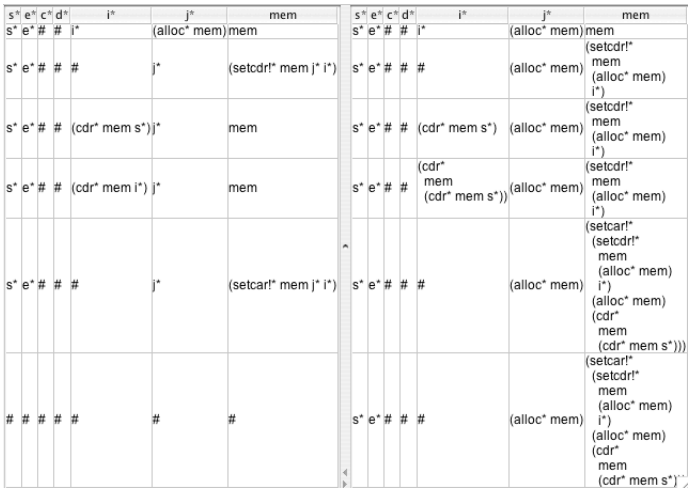


s*	e*	c*	d*	i*	j*	mem
s*	e*	#	#	i*	(alloc* mem)	mem
s*	e*	#	#	#	j*	(setcdr!* mem j* i*)
s*	e*	#	#	(cdr* mem s*)	j*	mem
s*	e*	#	#	(cdr* mem i*)	j*	mem
#	#	#	#	#	#	#

s*	e*	c*	d*	i*	j*	mem
s*	e*	#	#	i*	(alloc* mem)	mem
s*	e*	#	#	#	(alloc* mem)	(setcdr!* mem (alloc* mem) i*)
s*	e*	#	#	(cdr* mem s*)	(alloc* mem)	(setcdr!* mem (alloc* mem) i*)
s*	e*	#	#	(cdr* (setcdr!* mem (alloc* mem) i*) (cdr* mem s*))	(alloc* mem)	(setcdr!* mem (alloc* mem) i*)
s*	e*	#	#	#	(alloc* mem)	(setcar!* (setcdr!* mem (alloc* mem) i*) (alloc* mem) (cdr* mem (cdr* mem s*))



The last step in the retiming proposal produce a matching evaluation.



s*	e*	c*	d*	i*	j*	mem
s*	e*	#	#	i*	(alloc* mem)	mem
s*	e*	#	#	#	j*	(setcdr!* mem j* i*)
s*	e*	#	#	(cdr* mem s*)	j*	mem
s*	e*	#	#	(cdr* mem i*)	j*	mem
s*	e*	#	#	#	j*	(setcar!* mem j* i*)
#	#	#	#	#	#	#

s*	e*	c*	d*	i*	j*	mem
s*	e*	#	#	i*	(alloc* mem)	mem
s*	e*	#	#	#	(alloc* mem)	(setcdr!* mem (alloc* mem) i*)
s*	e*	#	#	(cdr* mem s*)	(alloc* mem)	(setcdr!* mem (alloc* mem) i*)
s*	e*	#	#	(cdr* mem (cdr* mem s*))	(alloc* mem)	(setcdr!* mem (alloc* mem) i*)
s*	e*	#	#	#	(alloc* mem)	(setcar!* (setcdr!* mem (alloc* mem) i*) (alloc* mem) (cdr* mem (cdr* mem s*)))
s*	e*	#	#	#	(alloc* mem)	(setcar!* (setcdr!* mem (alloc* mem) i*) (alloc* mem) (cdr* mem (cdr* mem s*))

Figure 38: Retiming of `ap-t` steps `17` through `19`. The key observation here is that `alloc*` produces a cell different from any `car*` or `cdr*` access to that heap.

into schedules. Adding behavior for *locate* would require manipulation in the DDD behavioral algebra. The `locate` function subsumes this control structure in SD. The schedule's prefix and suffix are the same in both WS and SD, relative to `locate` and its control loop.

As noted before, `rap` and `set` were dropped from HS because their behavior could not be expressed functionally with abstract pairs. Since SD operates on a heap model, it is a good starting point for explicitly specifying the behavior of side-effects. Unlike WS, HS does not model I/O—and neither does the SD system. As with side-effects, SD provides a good starting point for adding I/O capabilities.

Chapter 10

Conclusion

10.1 Achievements

Starfish extends the design derivation system with behavior tables, serialization tables, data refinement and retiming transformations. The fusion of these technologies have improved both the space of derivations and the ease of achieving them. Two case studies have shown that these techniques scale to medium-size systems with respect to the chosen level of abstraction—both the garbage collector and SECD machine expand into considerably larger descriptions at the gate-level.

Starfish implements *behavior table* representation, hierarchical connection, display and transformation algebra. Since behavior tables represent a compromise between behavioral and architectural views, they are ideally suited for imposing structural order on control-oriented specifications. Table rows align control actions while columns align component, or signal, actions. Action alignment is the root of perspicuity for system factorization. In an interactive setting, it informs allocation and binding for instructions, input signals and output signals.

The core algebra is sufficiently powerful to achieve *system factorization*, but their granularity impedes the interactive process. Starfish addresses this problem with

higher-level factorization tactics that manipulate factorization targets, inputs, and outputs. They provide a plausible starting point by automating binding and allocation for input signals. The designer can subsequently alter the automated results with property-oriented transformations (e.g., `permute-comb-signals`). Once the designer determines the input, output and target signals for a factorization, a higher-level decomposition command performs instruction assignment, table splitting, decision table reorganization, and useless signal elimination.

Even though behavior tables can illuminate factorization opportunities and paths to achieving them, the first proposal for their core algebra [61] matched the architectural transformations of its stream-equation predecessor. Starfish extends the algebra with *sequential identification* (p. 53)—a conversion method between combinational and sequential signals. The transformation enables *retiming*, which moves function evaluation to earlier or later steps. It is a primitive for manipulating pipelines. In addition to altering timing, sequential identification justifies, in part, Starfish's *data refinement* procedures, which replace abstract signals with implementation signals.

Starfish implements an explicit type system for behavior tables. This is a departure from DDD which checks type consistency with an appropriately defined abstract interpretation. Starfish adopts first-order multi-sorted term algebras as the basis for its type system. Designers can declare parameterized types, which overloads the type of symbols. An inferencer resolves most ambiguities, but designers can explicitly annotate subterms in case the algorithm fails. Type declarations optionally constrain their semantics with universally quantified identities. The collection of such identities forms a database for the *replacement rule*—term substitution justified by

algebraic equivalence. Starfish implements special declarative forms to support *data-refinement*. Data refinement results from sequential identification and homomorphic term identities. Starfish automates the term rewriting and signal transformations. A *stateful refinement* implements multiple abstract signals as references into a shared store. Starfish serializes access to the shared implementation state in case of simultaneous access at the abstract level.

Starfish implements interactive control serialization at the architectural level with *serialization tables*. DDD's behavioral algebra, rather than its architectural algebra, supports serialization. It is preferable to make the serialization decisions at the architectural level because the behavior table representation makes scheduling conflicts explicit. Serialization tables are a scheduling assistant that expands the evaluation of a single behavior-table action into a sequence. It displays intermediate actions and storage as well as their symbolic evaluation. Starfish only incorporates serializations whose symbolic interpretation equals the original specification terms. The designer can "argue" equivalence by applying term-level identities to the symbolic evaluations.

Starfish's extensions to the previous architectural algebra—retiming, data-refinement, and serialization—widen the space of decomposition. Two case studies illustrate this fact: a garbage collector decomposition and an SECD machine refinement. The first case study shows how data refinement and system factorization transform a system defined on abstract memory signals into a switched memory architecture. The second study shows how serialization, retiming, and data-refinement link a low-level SECD machine description to a high-level specification, similar to a textbook machine for operational semantics [42]. The original DDD-derivation of SECD hardware [103]

was not published because its starting specification anticipated too much of the target architecture. Starfish's SECD derivation establishes much of this missing link.

10.2 Future Work

Although Starfish makes several contributions to formal derivation, there are several open problems in the field. From the perspective of this thesis, some of the most evident directions for development include deriving interface protocols, accounting for timing changes between components, enriching behavior tables with bounded indirection, robustly scripting derivations, integrating with synchronous back-ends, and integrating with external formal verification tools. Finally, as a proof-of-concept prototype, its usability, speed, stability, and error-handling are all obvious points for improvement.

Transaction modeling

The transaction problem—how to communicate input and output values between architectural components—is invisible at the behavioral specification level and undefined in many architectural views. Derivation methodologies targeting low-level implementations need principled ways for introducing transaction protocols. In the context of DDD, Zhu [106] and, more generally, Rath [83] have proposed transaction derivation methods. Rath conceived his proposal, *Interface Specification Language* (ISL), as complementary to behavior tables. ISL expresses transactions as finite automata and their complements. Derivations incorporate transactions by directly inserting the automata actions into behavior tables—much like serializations

in Starfish expand single action over many steps. In simple cases, where the transaction consumes a fixed number of steps, serialization and data-refinement suffice for transaction introduction. In general though, transactions re-visit control states until an external condition has been satisfied; serialization does not capture this kind of protocol behavior.

Bounded Indirection

Just as ISL and behavior tables are complementary technologies, *bounded indirection* [98]—a specification mechanism for compactly expressing interrupts, continuations, and dynamic connections between machines—was proposed at the same time. An indirect type varies over the space of like-typed sequential signals. In addition to defining semantics for bounded indirection, the proposal introduces new transformations for manipulating behavior tables. Starfish, the first implementation of table-oriented derivation, is the natural platform for implementing bounded indirection, although the construct would benefit any hardware description language.

Accounting for alignment shifts

Serialization and interface protocol insertion alter input and output the timelines of their target components. Such timing changes can break communication assumptions with external components. Starfish side-steps this problem by only permitting serialization in flat behavior-table specifications—a choice that limits derivation flexibility. A full solution will account for the timing changes each transformation introduces, the timing policy or tolerance afforded by each component, and possibly the inter-component adjustment required to maintain correct communication.

Addressing and scripting

Robust sub-expression addressing is an important feature for developing derivations and an even more important feature for developing scripts. Fragile addressing schemes do not tolerate changes in derivation order without re-specifying expression references. Starfish uses guards and signal names specify sub-expressions in a behavior table. This alleviates some problems with stale sub-expression addressing when altering derivation scripts. However, addressing still relies on numeric branches and paths for specifying hierarchical nodes and subterms. Addressing would benefit from specifying paths by the user-defined names for system-node blocks rather than its position number in the tree representation. Similarly, subterm addressing in scripts could further benefit from a DDD-style s-expression specification [10], where subterms are specified by a sequence of function names.

Scripting in Starfish is limited to straight-line command evaluation. Some derivations consisting of tedious and predictable transformations were generated with user-defined Scheme functions. This works well enough for achieving known goals, but a scripting language with interfaces to the transformation engine and backtracking could explore derivation paths in pursuit of user-defined properties. For instance, known algorithms for high-level synthesis could guide the transformation process.

Analytical visualization

Visual feedback is one of the chief benefits of behavior tables. Starfish's support for subterm colorization is sparse. Expanding colorization to indicate transformation candidates, illustrate conflicting subexpressions from transformation errors, and reflect system properties (e.g., control-flow dependence) as the derivation proceeds would

greatly improve designer interaction. Often a transformation is invalid because of a conflicting term; for instance, a term instantiation could result in combinational feedback. Highlighting the dependency chain in the behavior table is a more informative way to report this error than a textual error.

More than just an error reporting device, colorization can also inform transformation decisions. Highlighting abstract types in a behavior table helps determine signal selection in a signal factorization. The retiming transformation seems particularly fruitful for visual analysis. In some cases, a retiming essentially moves function execution to a prior control state. If there are multiple ways to enter the source control state, the function may potentially be replicated in each of those previous states. A control-flow analysis which highlights the incoming and outgoing transitions can aid the retiming process.

Integration with external tools

Identities in Starfish's type and refinement declarations are user assertions. Moreover, Starfish does not specify term semantics beyond identity satisfaction. A formal workflow must specify term semantics and verify that the semantics satisfy the type and refinement declarations. The natural place to specify semantics is within DDD as functional Scheme expressions. Once specified, a theorem proving assistant such as PVS or Isabelle can validate the type structure and identities against the term semantics.

Although these interactions complete the formal justification for Starfish's results, Starfish still has no back-end for generating synchronous systems. If exported

to DDD, which an easy transformation, the stream interpreter can execute or simulate the architecture. The real payoff, however, comes from synchronous back-end compilers. Revitalizing low-level synthesis—the most recent interfaces are over a decade old—would bring the design methodology into the next stage of maturity by producing a testable implementation. Since Starfish expressions are independent of the synchronization mechanism, the synthesis target need not be hardware. Integrating a synchronous software back-end, such as the Giotto real-time compiler [45], is a promising option since embedded systems depend upon software as much or more than hardware. Giotto, in particular, aligns well with Starfish's principle of manual intervention since its compiler exploits user-defined annotations when generating its results.

Other systems, including DDD's commercial offspring DRS [13], provide a good model for how Starfish's role in the design process by successfully joining several tools together. DRS is the central tool in a workflow that integrates theorem provers, model checkers, equivalence checkers, and temporal logic specification with low-level tools for conventional simulation and synthesis. DRS performs a series of shallow embeddings to verify specification properties at a high level in PVS, and then model-check the results of their derivation and VHDL generator at a low-level. Conventional tools synthesize the last stages of the product; automated model extraction and verification maintain formal correctness.

Top-down design methods such as design derivation are natural candidates for organizing the application of external tool-sets. The long-term goal for design derivation—to which Starfish belongs—is to formalize the design process by delegating specification, synthesis, and verification tasks to external specialized tools in a coherent manner. Architectural decomposition, Starfish's task, lies at the heart of this coordination challenge.

Bibliography

[1] Mark Aagaard, Byron Cook, Nancy A. Day, and Robert B. Jones. A framework for microprocessor correctness statements. In *CHARME '01: Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 433–448, London, UK, 2001. Springer-Verlag.

[2] Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. Casl: The common algebraic specification language. *Theoretical Computer Science*, 286(2):153–196, 2002.

[3] Egidio Astesiano, H. J Kreowski, and B. Krieg-Bruckner, editors. *Algebraic Foundations of Systems Specification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[4] Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired: Wire-aware circuit design. In *CHARME*, pages 5–19, 2005.

[5] Jon Barwise and Lawrence Moss. *Vicious Circles*. CSLI Publications, 1996.

[6] G. Birtwistle and B. Graham. Verifying SECD in HOL. In J. Staunstrup, editor, *Formal Methods for VLSI Design: IFIP WG10.2*, Lecture Notes, pages 129–177. North-Holland, 1990.

[7] E. Boerger and J. K. Huggins. Abstract State Machines 1988-1998: Commented ASM Bibliography. *ArXiv Computer Science e-prints*, November 1998.

[8] Tommaso Bolognesi and Ed Brinksma. Introduction to the iso specification language lotos. *Comput. Netw. ISDN Syst.*, 14(1):25–59, 1987.

[9] B. Bose, M. E. Tuna, and J. M. Nagy. LavaCORE configurable Java Processor Core. In *2002 IEEE Aerospace Conference Proceedings*, volume 4, pages 4–1953–4–1959, 9-16 March 2002.

[10] Bhaskar Bose. DDD - a transformation system for digital design derivation. Technical Report 331, Indiana University Computer Science Department, May 1991.

[11] Bhaskar Bose. *DDD-FM9001: Derivation of a Verified Microprocessor*. PhD thesis, Computer Science Department, Indiana University, USA, 1994. Technical Report No. 456, 155 pages, //ftp.cs.indiana.edu/pub/techreports/TR456.html.

[12] Bhaskar Bose and Steven D. Johnson. DDD-FM9001: Derivation of a verified microprocessor. an exercise in integrating verification with formal derivation. In G. Milne and L. Pierre, editors, *Correct Hardware Design and Verification Methods (CHARME'93)*, volume 683 of *Lecture Notes in Computer Science*, pages 191–202. Springer, 1993. IFIP WG 10.2 Advanced Research Working Conference, CHARME'93, Arles, France, May 24-26, 1993, Proceedings.

[13] Bhaskar Bose, M. Esen Tuna, and Ingo Cyliax. FormalCORE^{TM} PCI/32 a formally verified VHDL synthesizable PCI core. In C. Michael Holloway, editor, *Lfm2000: Fifth NASA Langley Formal Methods Workshop*, pages 105–116, Langley Research Venter, Hampton, Virginia, June 2000.

[14] C.D. Boyer and Steven D. Johnson. Using the digital design derivation system: case study of a VLSI garbage collector. In Darringer and Ramming, editors, *Ninth International Symposium on Computer Hardware Description Languages (CHDL'89)*, Amsterdam, 1989. IFIP WG 10.2, Elsevier.

[15] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[16] Robert G. Burger. The scheme machine. Technical Report 413, Indiana University, Computer Science Department, August 1994. 59 pages.

[17] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM (JACM)*, 24(1):44–67, 1977.

[18] Rod M. Burstall and Joseph A. Goguen. The semantics of clear, a specification language. In *Proceedings of the Abstract Software Specifications, 1979 Copenhagen Winter School*, pages 292–332, London, UK, 1980. Springer-Verlag.

[19] Raul Camposano and Wayne H. Wolf, editors. *High-Level VLSI Synthesis*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[20] K. Claessen and M. Sheeran. A tutorial on lava: A hardware description and verification system, 2000.

[21] B. Cook, J. Launchbury, and J. Matthews. Specifying superscalar microprocessors in hawk, 1998.

[22] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.

[23] D. Eisenbiegler, R. Kumar, and C. Blumenrohr. A constructive approach towards correctness of synthesis-application within retiming. *edtc*, 00:427, 1997.

[24] Dirk Eisenbiegler and Ramayya Kumar. Formally embedding existing high level synthesis algorithms. In *CHARME '95: Proceedings of the IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 71–83, London, UK, 1995. Springer-Verlag.

[25] Hans Eveking, Holger Hinrichsen, and Gerd Ritter. Automatic verification of scheduling results in high-level synthesis. *date*, 00:59, 1999.

[26] Robert E. Filman and Daniel P. Friedman. *Coordinated computing: tools and techniques for distributed software*. McGraw-Hill, Inc., New York, NY, USA, 1984.

[27] Daniel P. Friedman, Christopher T. Haynes, and Mitchell Wand. *Essentials of programming languages (2nd ed.)*. Massachusetts Institute of Technology, Cambridge, MA, USA, 2001.

[28] Daniel P. Friedman and David S. Wise. Output driven interpretation of recursive programs, or writing creates and destroys data structures. *Information Processing Letters*, 5(6):155–160, 1976.

[29] Daniel P. Friedman and David S. Wise. Functional combination. *Comput. Lang.*, 3(1):31–35, 1978.

[30] Alfons Geser and Paul Miner. A formal correctness proof of the SPIDER diagnosis protocol. Theorem-Proving in Higher-Order Logics (TPHOLs), track B, 2002.

[31] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.

[32] Joseph A. Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Raymond Yeh, editor, *Current Trends in Programming Methodology*, pages 80–149. Prentice-Hall, 1978.

[33] Yuri Gurevich. Evolving algebras 1993: Lipari guide. pages 9–36, 1995.

[34] J V Guttag and J J Horning. Report on the larch shared language. *Sci. Comput. Program.*, 6(2):103–134, 1986.

[35] Joshua D. Guttman, John D. Ramsdell, and Mitchell Wand. VLISP: a verified implementation of Scheme. *Lisp and Symbolic Computation*, 8:5–32, 1995.

[36] D. Harel. Statecharts: a visual formalism for complex systems. *The Science of Computer Programming*, 8:231–274, 1987.

[37] David Harel. On visual formalisms. *Commun. ACM*, 31(5):514–530, 1988.

[38] Mats P.E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, 1996.

[39] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: a toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS '95)*, pages 109–122, 1995.

[40] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.*, 5(3):231–261, 1996.

[41] Constance L. Heitmeyer, James Kirby, Bruce G. Labaw, and Ramesh Bharadwaj. Scr*: A toolset for specifying and analyzing software requirements. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 526–531, London, UK, 1998. Springer-Verlag.

[42] Peter Henderson. *Functional Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1980.

[43] K.L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions Software Engineering*, SE-6:2–13, 1980.

[44] K.L. Heninger, J. Kallander, D.L. Parnas, and J.E. Shore. Software requirements for the A-7 aircraft. NRL Memorandum Report 3876, United States Naval Research Laboratory, Washington, D.C., November 1978.

[45] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: a time-triggered language for embedded programming. Technical report, Berkeley, CA, USA, 2001.

[46] C. A. R. Hoare. Proof of correctness of data representation. In *Language Hierarchies and Interfaces, International Summer School*, pages 183–193, London, UK, 1976. Springer-Verlag.

[47] D. N. Hoover and Zewei Chen. Tbell: A mathematical tool for analyzing decision tables. Contractor Report 195027, National Aeronautics and Space Administration, Hampton VA 23681-0001, November 1994. Authors' affiliation: Odyssey Research Associates, Inc., Ithaca NY.

[48] D. N. Hoover and Zewei Chen. Tablewise, a decision table tool. In *Computer Assurance, 1995. COMPASS '95. 'Systems Integrity, Software Safety and Process Security'. Proceedings of the Tenth Annual Conference on Computer Assurance*, pages 97–108, June 1995.

[49] D. N. Hoover, David Guaspari, and Polar Humenn. Applications of formal methods to specification and safety of avionics software. Contractor Report 4723, National Aeronautics and Space Administration Langley Research Center (NASA/LRC), Hampton VA 23681-0001, November 1994. Authors affiliation: Odyssey Research Associates, Inc., Ithaca NY. Printed copies available from NASA Center for AeroSpace Information, 800 Elkridge Landing Road, Linthicum Heights MD 21090-2934.

[50] Steven D. Johnson. An interpretive model for a language based on suspended construction. Technical Report 68, Indiana University Computer Science Department, Bloomington, Indiana, Aug 1977.

[51] Steven D. Johnson. Applicative programming and digital design. In *Proc. Eleventh Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming (POPL'84)*, pages 218–227, 1984.

[52] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations.* MIT Press, Cambridge, 1984.

[53] Steven D. Johnson. Digital design in a functional calculus. In G. Milne and P.A. Subramanyam, editors, *Formal Aspects of VLSI Design*, pages 153–178. North-Holland, Amsterdam, 1986. Proceedings of tthe 1985 Edingurgh Workshop on VLSI, Edinburgh, Scotl and, U.K.

[54] Steven D. Johnson. Manipulating logical organization with system factorizations. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *Lecture Notes in Computer Science*, pages 260–281. Springer, July 1989. Mathematical Sciences Institute Workshop, Cornell University, Ithaca, New York, USA, July 1989, Proceedings.

[55] Steven D. Johnson. A tabular language for system design. In C. Michael Holloway and Kelly J. Hayhurst, editors, *Lfm97: Fourth NASA Langley Formal Methods Workshop*, September 1997. NASA Conference Publication 3356, Proceedings of the 4th NASA Langley Formal Methods Workshop, Hampton, Virginia 10-12 September, 1997, http://archive.larc.nasa.gov/shemesh/Lfm97/.

[56] Steven D. Johnson. Formal derivation of a scheme computer. Technical Report 544, Indiana University Computer Science Dept., Bloomington, Indiana, September 2000. http://www.cs.indiana.edu/ftp/techreports/.

[57] Steven D. Johnson. View from the fringe of the fringe. In Tiziana Margaria and Thomas Melham, editors, *11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods, CHARME 2001, Livingston, Scotland, Proceedings*, volume 2144 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 2001. Invited paper.

[58] Steven D. Johnson, B. Bose, and C.D. Boyer. A tactical framework for digital design. In Birtwistle and Subramanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 349–383. Kluwer, Boston, 1988.

[59] Steven D. Johnson and Bhaskar Bose. A system for mechanized digital design derivation. In *IFIP and ACM/SIGDA International Workshop on Formal Methods in VLSI Design*, 1991. IFIP WG 10.2 conference series. Available as Indiana University Computer Science Department Technical Report No. 323 (rev. 1997).

[60] Steven D. Johnson and C.D. Boyer. Modelling transistors applicatively. In G.J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 397–420. North-Holland, 1988. Proceedings of the IFIP WG 10.2 Working Conference on The Fusion of Hardware Design and Vierification, Glasgow, Scotland, 4-6 July, 1988.

[61] Steven D. Johnson and Alex Tsow. Algebra of behavior tables. In C. M. Holloway, editor, *Lfm2000: Fifth NASA Langley Formal Methods Workshop*, pages 23–34, 2000. NASA Conference Publication NASA/CP-2000-210100. Proceedings of the 5th NASA Langley Formal Methods Workshop, Williamsburg, Virginia, 13-15 June, 2000, http://shemesh.larc.nasa.gov/fm/Lfm2000/.

[62] Steven D. Johnson, R.M. Wehrmeister, and B. Bose. On the interplay of synthesis and verification: Experiments with the FM8501 processor description. In Claesen, editor, *Formal VLSI Specification and Synthesis, VLSI Design Methods - I*, pages 385–404. North-Holland, 1990. Proceedings of the IFIP WG 10.2/WG 10.5 International Worksho p on Applied Formal Methods for Correct VlSI Design, Sponsored by IMEC, Houtha len, Belgium, 13-16 November, 1989.

[63] Peter J. Landin. The mechanical evaluation of expressions. 6(4):308–320, January 1964.

[64] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specifiation for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[65] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[66] John Matthews and John Launchbury. Elementary microarchitecture algebra. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 288–300, London, UK, 1999. Springer-Verlag.

[67] David Megginson. Megginson Technologies: Simple API for XML. Accessed 18 June 2007, `http://www.megginson.com/downloads/SAX/`.

[68] J Meseguer and J A Goguen. Initiality, induction, and computability. pages 459–541, 1986.

[69] José Meseguer. A logical theory of concurrent objects and its realization in the maude language. pages 314–390, 1993.

[70] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1994.

[71] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[72] Paul S. Miner. *Hardware Verification using Coinductive Assertions*. PhD thesis, Computer Science Department, Indiana University, USA, June 1998. Technical Report No. 510, 138 pages.

[73] Paul S. Miner and Steven D. Johnson. Verification of an optimized fault-tolerant clock synchronization circuit. In Mary Sheeran and Satnam Singh, editors, *Designing Correct Circuits*, Electronic Workshops in Computing, http://www.ewic.org.uk/ewic/workshop/view.cfm/DCC-96. Springer, September 1996.

[74] Lawrence S. Moss. Parametric corecursion. *Theoretical Computer Science*, 260(1-2):139–163, 2001.

[75] John O'Donnell. HYDRA: hardware description in a functional language using recursion equations and high order combining forms. In G.J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 309–328. North-Holland, 1988. Proceedings of the IFIP WG 10.2 Working Conference on The Fusion of Hardware Design and Vierification, Glasgow, Scotland, 4-6 July, 1988.

[76] John O'Donnell. Overview of hydra: A concurrent language for synchronous digital circuit design. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 240, Washington, DC, USA, 2002. IEEE Computer Society.

[77] Sam Owre, John M. Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *CADE-11: Proceedings of the 11th International Conference on Automated Deduction*, pages 748–752, London, UK, 1992. Springer-Verlag.

[78] D. L. Parnas. Tabular representation of relations. Technical Report 260, Communications Research Laboratory, McMaster University, 1992.

[79] David Lorge Parnas. Inspection of safety-critical software using program-function tables. In *IFIP Congress (3)*, pages 270–277, 1994.

[80] David Lorge Parnas. Some theorems we should prove. In *HUG '93: Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 155–162, London, UK, 1994. Springer-Verlag.

[81] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[82] R. Radhakrishnan, E. Teica, and R. Vermuri. An approach to high-level synthesis system validation using formally verified transformations. *hldvt*, 00:80, 2000.

[83] Kamlesh Rath. *Sequential System Decomposition*. PhD thesis, Computer Science Department, Indiana University, USA, 1995. Technical Report No. 457, 90 pages.

[84] Kamlesh Rath, Venkatesh Choppella, and Steven D. Johnson. Decomposition of sequential behavior using interface specification and complementation. *VLSI Design*, 3(3-4):347–358, 1995.

[85] Kamlesh Rath and Steven D. Johnson. Toward a basis for protocol specification and process decomposition. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications (CHDL'93)*, volume A-32 of *IFIP Transactions*, pages 169–186. North-Holland, 1993. Proceedings of the 11th IFIP WG 10.2 International Conference on Computer Hardware Description Languages and their Applications - CHDL'93, sponsored by IFIP WG 10.2 and incooperation with IEEE COMPSOC, Ottawa, Ontario, Canada, 26-28 April, 1993.

[86] Kamlesh Rath, M. Esen Tuna, and Steven D. Johnson. Behavior tables: A basis for system representation and transformational system synthesis. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD'93)*, pages 736–740. IEEE Cat. No. 93CH3344-9, November 1993.

[87] Markus Roggenbach. Csp-casl: a new integration of process algebra and algebraic specification. *Theor. Comput. Sci.*, 354(1):42–71, 2006.

[88] David Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):175–234, 1996.

[89] Donald Sannella. Formal program development in Extended ML for the working programmer. In *Proc. 3rd BCS/FACS Workshop on Refinement*, Workshops in Computing, pages 99–130. Springer, 1991.

[90] Donald Sannella. Algebraic specification and program development by stepwise refinement. In *Proc. 9th Intl. Workshop on Logic-based Program Synthesis and Transformation, LOPSTR'99*, volume 1817 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2000.

[91] Mary Sheeran. Generating fast multipliers using clever circuits.

[92] Mary Sheeran. Retiming and slowdown in ruby. In G.J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 289–308. North-Holland, 1988. Proceedings of the IFIP WG 10.2 Working Conference on The Fusion of Hardware Design and Vierification, Glasgow, Scotland, 4-6 July, 1988.

[93] Daniel J. Sorin, Manoj Plakal, Anne E. Condon, Mark D. Hill, Milo M.K. Martin, and David A. Wood. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):556–578, 2002.

[94] Robert F. Stark, Joachim Schmid, and E. Borger. *Java and the Java Virtual Machine: Definition, Verification, Validation with CDrom.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[95] T. F. Melham. Abstraction mechanisms for hardware verification. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 129–157, Boston, 1988. Kluwer Academic Publishers.

[96] Alex Tsow and Steven D. Johnson. Visualizing system factorizations with behavior tables. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, TX, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 523–541, Heidelberg Berlin, 2000. Springer-Verlag.

[97] Alex Tsow and Steven D. Johnson. Data refinement for synchronous system specification and construction. In Dominique Borrione and Wolfgang Paul, editors, *Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *Lecture Notes in Computer Science*, pages 398–401, Berlin, October 2005. Springer-Verlag. 13th IFIP WG 10.5 Advanced Research Working Conference.

[98] M. Esen Tuna, Kamlesh Rath, and Steven D. Johnson. Specification and synthesis of bounded indirection. In *Proceedings of the Fifth Great Lakes Symposium on VLSI (GLSVLS I'95)*, pages 86–89. IEEE, March 1995.

[99] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27:164–180, 1980.

[100] Mitchell Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, July 1982.

[101] Mitchell Wand. Semantics-directed machine architecture. In *Proceedings 9th ACM Symposium on Programming Languages*, pages 234–241, 1982.

[102] Mitchell Wand and Daniel P. Friedman. Compiling lambda expressions using continuations and factorizations. *Journal of Computer Languages*, 3:241–263, 1978.

[103] R.M. Wehrmeister. Derivation of an SECD machine: Experience with a transformational approach to synthesis. Technical Report 290, Indiana University, Computer Science Department, September 1989.

[104] David E. Winkel and Franklin P. Prosser. *Art of Digital Design: An Introduction to Top-Down Design*. Prentice Hall Professional Technical Reference, 1987.

[105] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, 1971.

[106] Zheng Zhu. *Structured Hardware Design Transformations*. PhD thesis, Computer Science Department, Indiana University, USA, 1992.

[107] Zheng Zhu and Steven D. Johnson. An example of interactive hardware transformation. In P. A. Subramanyam, editor, *IFIP and ACM/SIGDA International Workshop on Formal Methods in VLSI Design*, January 1991. IFIP WG 10.2 conference series.

Appendix A

SECD derivation details

SECD specification file

```
(define-enum-alg secd-state
  (rtn-t dum-t ap-t sel-t join-t car-t cdr-t cons-t ld-t ldc-t ldf-t atom-t eq-t
    leq-t add-t sub-t sl-t ls-t ci-t num-t sym-t pair-t exec-t pop-t stop-t)
  ()
  ()
  ())


(define-term-alg lt
  (nil err tru fls)
  ((cons 2) (car 1) (cdr 1)
   (nil? 1)
   (locate 2)
   (atom? 1)
   (eq? 2)
   (<=? 2)
   (plus 2)
   (minus 2)
   (test? 2)
   (if 3)
   (sym-list 1)
   (list-sym 1)
   (char2int 1)
   (num? 1)
   (pair? 1)
   (sym? 1))
  (key a b)
  (('access1 (car (cons a b)) a)
   ('access2 (cdr (cons a b)) b)
;   ('pair-tst (pair? (cons a b)) tru)
;   ('nil-tst1 (nil? nil) tru)
;   ('nil-tst2 (nil? tru) fls)
;   ('nil-tst3 (nil? fls) fls)
;   ('nil-tst4 (nil? (cons a b)) fls)
   ))


(declare-funcs secd-helpers
  ()
  ([dcd ("lt") "secd-state"]
   [test-inst? ("secd-state" "lt") "lt"]
;   [true?* ("lt") "boolean"]
   [true? ("lt") "boolean"]
   )
  ([a "lt"] [b "lt"] [k "lt"] [tst "boolean"])
  (['if-def "(if k a b)" "(sel (true? k) a b)"]
   ['test?-def "(test? a b)" "(test-inst? (dcd a) b)"]
```

```
   ['test-inst?-def-atom "(test-inst? atom-t b)" "(atom? b)"]
   ['test-inst?-def-num "(test-inst? num-t b)" "(num? b)"]
   ['test-inst?-def-sym "(test-inst? sym-t b)" "(sym? b)"]
   ['test-inst?-def-pair "(test-inst? pair-t b)" "(pair? b)"]
;   ['assert1 "(sel tst (car a) (car b))" "(car (sel tst a b))"]
   ))


(define-param-alg lmem
  (cell)
  ()
  ([car* (lmem cell) cell]
   [cdr* (lmem cell) cell]
   [alloc* (lmem) cell]
   [setcar!* (lmem cell cell) lmem]
   [setcdr!* (lmem cell cell) lmem])
  ([m lmem] [i cell] [j cell] [k cell] [l cell])
  (['car*-setcdr!* (car* (setcdr!* m j k) i) (car* m i)]
   ['cdr*-setcar!* (cdr* (setcar!* m j k) i) (cdr* m i)]
   ['setcar!*-setcdr!*-comm
     (setcar!* (setcdr!* m i j) k l)
     (setcdr!* (setcar!* m k l) i j)]
   ))


; want, but cannot express general commutativity
; between setcar!* and setcdr!*

(define-term-alg lc
  (nil* err* fls* tru*)
  ((nil?* 1)
   (atom?* 1)
   (eq?* 2)
   (<=?* 2)
   (test?* 2)
   (plus* 2)
   (minus* 2)
   (sym-list* 1)
;   (list-sym* 1)
   (char2int* 1)
   (num?* 1)
   (pair?* 1)
   (sym?* 1))
  ()
  ())

;; Expresses specific features of lmem{lc}

(declare-funcs heap-helpers
  ()
  ([true?* ("lc") "boolean"]
   [lc-eq? ("lc" "lc") "boolean"]
   [locate* ("lmem{lc}" "lc" "lc") "lc"]
   [list-sym* ("lmem{lc}" "lc") "lc"])
  ([m "lmem{lc}"] [i "lc"] [j "lc"] [k "lc"])
  (['setcdr!*-inits-car-to-nil
     "(setcdr!* m (alloc* m) i)"
     "(setcar!* (setcdr!* m (alloc* m) i) (alloc* m) nil*)"]
   ['setcar!*-inits-cdr-to-nil
     "(setcar!* m (alloc* m) i)"
     "(setcdr!* (setcar!* m (alloc* m) i) (alloc* m) nil*)"]
```

```
   ['setcdr!*-effects-mem
     "(cdr* (setcdr!* m i j) k)"
     "(sel (lc-eq? i k) j (cdr* m k))"]

   ['setcar!*-effects-mem
     "(car* (setcar!* m i j) k)"
     "(sel (lc-eq? i k) j (car* m k))"]

   ['lc-eq?-comm "(lc-eq? i j)" "(lc-eq? j i)"]
   ['lc-eq?-true "(lc-eq? i i)" "true"]
   ['lc-eq?-false-1 "(lc-eq? (alloc* m) (car* m i))" "false"]
   ['lc-eq?-false-2 "(lc-eq? (alloc* m) (cdr* m i))" "false"]

   ))


(declare-refinement pr => cell
  ();; no parameters for now
  "lt"
  "lc"
  ([m "lmem{lc}"
     "#:lmem{lc}"])
  ([a "lt"]
   [b "lt"]
   [i "lc"]
   [j "lc"]
   [tst "boolean"]
   )

  ;; An unspecified refinement results in (<name> <state-symbol> <src-val>)

  (
    ["#:lt" "#:lc" "m"]
    ["(cons a b)"
     ; Basically "(alloc* m)" with call by value threading of state
     "(alloc* (pr=>cell.m a
               (pr=>cell.m b m)))"

     "(setcar!* (setcdr!* (pr=>cell.m a (pr=>cell.m b m))
                  (alloc* (pr=>cell.m a (pr=>cell.m b m)))
                  (pr=>cell b m))
        (alloc* (pr=>cell.m a (pr=>cell.m b m)))
        (pr=>cell a (pr=>cell.m b m)))"
     ]

    ["(car a)"
     "(car* (pr=>cell.m a m) (pr=>cell a m))"
     "(pr=>cell.m a m)"
     ]

    ["(cdr a)"
     "(cdr* (pr=>cell.m a m) (pr=>cell a m))"
     "(pr=>cell.m a m)"
     ]

    ;; The tag-checking lc primitives will be extended ALU ops
;     ((nil?* 1)
;      (atom?* 1)
;      (eq?* 2)
;      (<=?* 2)
```

```
;      (plus* 2)
;      (minus* 2)
;      (sym-list* 1)
;      (list-sym* 1)
;      (char2int* 1)
;      (num?* 1)
;      (pair?* 1)
;      (sym?* 1))


    ["(nil? a)"
     "(nil?* (pr=>cell a m))"
     "(pr=>cell.m a m)"]
    ["(atom? a)"
     "(atom?* (pr=>cell a m))"
     "(pr=>cell.m a m)"]
    ["(num? a)"
     "(num?* (pr=>cell a m))"
     "(pr=>cell.m a m)"]
    ["(pair? a)"
     "(pair?* (pr=>cell a m))"
     "(pr=>cell.m a m)"]
    ["(sym? a)"
     "(sym?* (pr=>cell a m))"
     "(pr=>cell.m a m)"]
    ["(test? a b)"
     "(test?* (pr=>cell a (pr=>cell.m b m)) (pr=>cell b m))"
     "(pr=>cell.m a (pr=>cell.m b m))"]
; Straight ALU ops
    ["(eq? a b)"
     "(eq?* (pr=>cell a (pr=>cell.m b m)) (pr=>cell b m))"
     "(pr=>cell.m a (pr=>cell.m b m))"]
    ["(<=? a b)"
     "(<=?* (pr=>cell a (pr=>cell.m b m)) (pr=>cell b m))"
     "(pr=>cell.m a (pr=>cell.m b m))"]
; Straight ALU ops
; Integers have a uniform representation (e.g. hashed)
; and do not actually reference the memory in a non trivial way.
    ["(plus a b)"
     "(plus* (pr=>cell a (pr=>cell.m b m)) (pr=>cell b m))"
     "(pr=>cell.m a (pr=>cell.m b m))"]
    ["(minus a b)"
     "(minus* (pr=>cell a (pr=>cell.m b m)) (pr=>cell b m))"
     "(pr=>cell.m a (pr=>cell.m b m))"]
;; Chars are also hashed as integers and require
;; no memory updates
    ["(char2int a)"
     "(char2int* (pr=>cell a m))"
     "(pr=>cell.m a m)"]
;; This is just a tag change in the expected model
    ["(sym-list a)"
     "(sym-list* (pr=>cell a m))"
     "(pr=>cell.m a m)"]
;; This is a tag change plus a well formedness check in the expected model.
;; Accesses but does not change the heap for the well formedness.
    ["(list-sym a)"
     "(list-sym* (pr=>cell.m a m) (pr=>cell a m))"
     "(pr=>cell.m a m)"]
;; Locate is equivalent to a couple of list-refs. It is a recursive
;; accessor that has no effect on the memory.  We do not define its
```

```
;; behavior here though.  It would be nice to say the value maps to
;; a one-level unrolling of its recursive definition
;     ["(locate a b)"
;      "(pr=>cell (locate a b) m)"
;      "(pr=>cell.m a (pr=>cell.m b m))"]
    ["(locate a b)"
     "(locate* (pr=>cell.m a (pr=>cell.m b m)) (pr=>cell a (pr=>cell.m b m)) (pr=>cell b m))"
     "(pr=>cell.m a (pr=>cell.m b m))"]

; Punting on tag changing ops. These should update the memory

    ["nil" "nil*" "m"]
    ["fls" "fls*" "m"]
    ["tru" "tru*" "m"]
    ["err" "err*" "m"]
   )

  ;; For functions that consume lt's, but do not produce lt's?
;  (['pr=>cell:true?*
;     "(true?* a)"
;     "(inv (lc-eq? (pr=>cell a m) fls*))"])
  (['pr=>cell:true?
     "(true? a)"
     "(true?* (pr=>cell a m))"])

  ;;  abstraction identities, this should be the usual ident-dt
  (
    ['pr=>cell-inv
      "(pr<=cell (pr=>cell a m)
         (pr=>cell.m a m))" "a"]
    ;; No cons-decode here :(
    ;; This may require more sophisticated cells.
    ;; [tag addr] may be appropriate.
    ['pr<=cell-nil-decode "(pr<=cell nil* m)" "nil*"]
    ['pr<=cell-fls-decode "(pr<=cell fls* m)" "fls*"]
    ['pr<=cell-tru-decode "(pr<=cell tru* m)" "tru*"]
    ['pr<=cell-err-decode "(pr<=cell err* m)" "err*"]

    ['assert2 "(sel (inv tst) i j)" "(sel tst j i)"]

 )
)


;; Table spec

(define-sys-table secd
  ([none "lt"]) ; for now, no inputs
  ([s "lt" seq] [e "lt" seq] [c "lt" seq] [d "lt" seq])
  ()
  (none)
  ("(dcd (car c))")
  ((["rtn-t"] ["(cons (car s) (car d))" "(car (cdr d))" "(car (cdr (cdr d)))" "(cdr (cdr (cdr d)))"])
   (["dum-t"] ["s" "(cons nil e)" "(cdr c)" "d"])
   (["ap-t"]  ["nil" "(cons (car (cdr s)) (cdr (car s)))" "(car (car s))"
                     "(cons (cdr (cdr s)) (cons e (cons (cdr c) d)))"])
   ; cannot define setcar! until data transformation
   ; (("rap") (nil (setcar! (cdr (car s)) (car (cdr s)))
   ;           (car (car s)) (cons (cdr (cdr s)) (cons (cdr e) (cons (cdr c) d)))))
   (["sel-t"] ["(cdr s)" "e" "(if (car s) (car (cdr c)) (car (cdr (cdr c))))"
               "(cons (cdr (cdr (cdr c))) d)"])
```

```
   (["join-t"] ["s" "e" "(car d)" "(cdr d)"])
   ;; Loading variables, functions, and constants onto the stack
   (["ld-t"] ["(cons (locate (car (cdr c)) e) s)" "e" "(cdr (cdr c))" "d"])
   (["ldc-t"] ["(cons (car (cdr c)) s)" "e" "(cdr (cdr c))" "d"])
   (["ldf-t"] ["(cons (cons (car (cdr c)) e) s)" "e" "(cdr (cdr c))" "d"])
;   (("rech") ((cons (readchar) s) e (cdr c) d))
;   (("wrch") (writechar (car s)) (s e (cdr c) d))
   (["exec-t"] ["(cons (cons (car s) e) (cdr s))" "e" "(cdr c)" "d"])
   (["pop-t"] ["(cdr s)" "e" "(cdr c)" "d"])
;   (("set") (cons (set (car (cdr c)) e (car s)) (cdr s)) e (cdr (cdr c)) (d))
   ;; List ops
   (["car-t"] ["(cons (car (car s)) (cdr s))" "e" "(cdr c)" "d"])
   (["cdr-t"] ["(cons (cdr (car s)) (cdr s))" "e" "(cdr c)" "d"])
   (["cons-t"] ["(cons (cons (car s) (car (cdr s))) (cdr (cdr s)))" "e" "(cdr c)" "d"])
   ;; These are the constant time binary ops
   (["eq-t"] ["(cons (eq? (car s) (car (cdr s))) (cdr (cdr s)))" "e" "(cdr c)" "d"])
   (["leq-t"] ["(cons (<=? (car s) (car (cdr s))) (cdr (cdr s)))" "e" "(cdr c)" "d"])
   (["add-t"] ["(cons (plus (car s) (car (cdr s))) (cdr (cdr s)))" "e" "(cdr c)" "d"])
   (["sub-t"] ["(cons (minus (car s) (car (cdr s))) (cdr (cdr s)))" "e" "(cdr c)" "d"])
   ;; These are the constant time unary ops
   (["atom-t"] ["(cons (atom? (car s)) (cdr s))" "e" "(cdr c)" "d"])
   (["num-t"] ["(cons (num? (car s)) (cdr s))" "e" "(cdr c)" "d"])
   (["sym-t"] ["(cons (sym? (car s)) (cdr s))" "e" "(cdr c)" "d"])
   (["pair-t"] ["(cons (pair? (car s)) (cdr s))" "e" "(cdr c)" "d"])
   ;; These are the recursive multi-step built in funcitons
   (["sl-t"] ["(cons (sym-list (car s)) (cdr s))" "e" "(cdr c)" "d"])
   (["ls-t"] ["(cons (list-sym (car s)) (cdr s))" "e" "(cdr c)" "d"])
   (["ci-t"] ["(cons (char2int (car s)) (cdr s))" "e" "(cdr c)" "d"])
   ;; Halt
   (["stop-t"] ["s" "e" "c" "d"])
   ))

secd
```

SECD derivation file

```
(load-spec "examples/secd/secd.ss")
(begin-serialization (make-sys-path) (quote ("rtn-t")) (quote ((s . "#") (e . "#") (c . "#") d)))
(ser-insert-col (quote i) "lt")
(ser-insert-col (quote j) "lt")
(ser-set-cell 4 0 "(car s)")
(new-ser-row (quote j) "(car d)")
(new-ser-row (quote ((s . "(cons i j)") (i . "#:lt") (j . "#:lt"))))
(new-ser-row (quote j) "(cdr d)")
(new-ser-row (quote i) "(car j)")
(new-ser-row (quote j) "(cdr j)")
(new-ser-row (quote ((e . "i") (i . "#:lt"))))
(new-ser-row (quote i) "(car j)")
(new-ser-row (quote ((c . "i") (i . "#:lt"))))
(new-ser-row (quote ((i . "(cdr j)") (j . "#:lt"))))
(new-ser-row (quote ((d . "i") (i . "#:lt"))))
(insert-ser-tab)
(begin-serialization (make-sys-path) (quote ("#" "dum-t"))
                                     (quote (s e (c . "#") d i (j . "(cdr c)"))))
(new-ser-row (quote ((e . "#:lt") (i . "e"))))
(new-ser-row (quote ((j . "#:lt") (c . "j"))))
(new-ser-row (quote ((j . "(cons nil i)") (i . "#:lt"))))
(new-ser-row (quote ((j . "#:lt") (e . "j"))))
(insert-ser-tab)
(begin-serialization (make-sys-path) (quote ("#" "ap-t")) (quote (s e c d i (j . "(car s)"))))
```

```
(new-ser-row (quote j) "(cdr j)")
(new-ser-row (quote i) "(cdr s)")
(new-ser-row (quote i) "(car i)")
(new-ser-row (quote i) "(cons i j)")
(new-ser-row (quote ((e . "#:lt") (j . "e"))))
(new-ser-row (quote ((e . "i") (i . "#:lt"))))
(new-ser-row (quote ((i . "(cdr c)") (c . "#:lt"))))
(new-ser-row (quote ((c . "j") (j . "#:lt"))))
(new-ser-row (quote ((d . "#:lt") (i . "#:lt") (j . "(cons i d)"))))
(new-ser-row (quote ((c . "#:lt") (j . "#:lt") (i . "(cons c j)"))))
(new-ser-row (quote j) "(cdr s)")
(new-ser-row (quote j) "(cdr j)")
(new-ser-row (quote ((j . "(cons j i)") (i . "#lt"))))
(new-ser-row (quote ((d . "j") (j . "#lt"))))
(new-ser-row (quote ((i . "(car s)") (s . "#lt"))))
(new-ser-row (quote i) "(car i)")
(new-ser-row (quote ((c . "i") (i . "#lt"))))
(new-ser-row (quote s) "nil")
(insert-ser-tab)
(apply-alg-ident (quote ()) (quote ("#" "sel-t")) "c" (quote if-def))
(apply-alg-ident (quote ()) (quote ("#" "sel-t")) "c" (quote assert1))
(begin-serialization (make-sys-path) (quote ("#" "sel-t")) (quote (s e c d (i . "(car s)") j)))
(new-ser-row (quote j) "(cdr c)")
(new-ser-row (quote ((i . "#:lt") (j . "(sel (true?* i) j (cdr j))"))))
(new-ser-row (quote j) "(car j)")
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote ((c . "j") (j . "#:lt"))))
(new-ser-row (quote i) "(cdr i)")
(new-ser-row (quote i) "(cdr i)")
(new-ser-row (quote ((d . "#:lt") (i . "#:lt") (j . "(cons i d)"))))
(new-ser-row (quote ((j . "#:lt") (d . "j"))))
(new-ser-row (quote ((s . "#:lt") (i . "(cdr s)"))))
(new-ser-row (quote ((i . "#:lt") (s . "i"))))
(insert-ser-tab)
(begin-serialization (make-sys-path) (quote ("#" "join-t"))
                                     (quote (s e (c . "#") d (i . "(car d)") j)))
(new-ser-row (quote ((c . "i") (i . "#:lt"))))
(new-ser-row (quote ((d . "#:lt") (i . "(cdr d)"))))
(new-ser-row (quote ((d . "i") (i . "#:lt"))))
(insert-ser-tab)
(begin-serialization (make-sys-path) (quote ("#" "car-t")) (quote (s e c d (i . "(car s)") j)))
(new-ser-row (quote i) "(car i)")
(new-ser-row (quote ((s . "#:lt") (j . "(cdr s)"))))
(new-ser-row (quote ((i . "#:lt") (j . "(cons i j)"))))
(new-ser-row (quote ((j . "#:lt") (s . "j"))))
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote ((i . "#:lt") (c . "i"))))
(insert-ser-tab)
(begin-serialization (make-sys-path) (quote ("#" "cdr-t")) (quote (s e c d (i . "(car s)") j)))
(new-ser-row (quote i) "(cdr i)")
(new-ser-row (quote ((s . "#:lt") (j . "(cdr s)"))))
(new-ser-row (quote ((i . "#:lt") (j . "(cons i j)"))))
(new-ser-row (quote ((j . "#:lt") (s . "j"))))
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote ((i . "#:lt") (c . "i"))))
(insert-ser-tab)
(begin-serialization (make-sys-path) (quote ("#" "ld-t")) (quote (s e c d (i . "(cdr c)") j)))
(new-ser-row (quote j) "e")
(new-ser-row (quote i) "(car i)")
(new-ser-row (quote ((i . "(locate i j)") (j . "#:lt"))))
(new-ser-row (quote ((j . "(cons i s)") (i . "#:lt") (s . "#:lt"))))
```

```
(new-ser-row (quote ((s . "j") (j . "#:lt"))))
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote i) "(cdr i)")
(new-ser-row (quote ((i . "#:lt") (c . "i"))))
(insert-ser-tab)
(begin-serialization (make-sys-path)
                     (quote ("#" "ldc-t")) (quote (s e (c . "#") d (i . "(cdr c)") j)))
(new-ser-row (quote c) "i")
(new-ser-row (quote i) "(car i)")
(new-ser-row (quote ((j . "(cons i s)") (i . "#:lt") (s . "#:lt"))))
(new-ser-row (quote ((s . "j") (j . "#:lt"))))
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote ((i . "#:lt") (c . "i"))))
(insert-ser-tab)
(begin-serialization (make-sys-path)
                     (quote ("#" "ldf-t")) (quote (s e (c . "#") d (i . "(cdr c)") j)))
(new-ser-row (quote c) "i")
(new-ser-row (quote i) "(car i)")
(new-ser-row (quote ((j . "(cons i e)") (i . "#:lt"))))
(new-ser-row (quote ((i . "(cons j s)") (j . "#:lt") (s . "#:lt"))))
(new-ser-row (quote ((s . "i") (i . "#:lt"))))
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote ((i . "#:lt") (c . "i"))))
(insert-ser-tab)
(begin-serialization (make-sys-path) (quote ("#" "exec-t")) (quote (s e c d (i . "(car s)") j)))
(new-ser-row (quote ((i . "#:lt") (j . "(cons i e)"))))
(new-ser-row (quote ((s . "#:lt") (i . "(cdr s)"))))
(new-ser-row (quote ((i . "#:lt") (j . "#:lt") (s . "(cons j i)"))))
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote ((i . "#:lt") (c . "i"))))
(insert-ser-tab)
(begin-serialization (make-sys-path)
                     (quote ("#" "pop-t")) (quote ((s . "#") e c d (i . "(cdr s)") j)))
(new-ser-row (quote ((i . "#:lt") (s . "i"))))
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote ((i . "#:lt") (c . "i"))))
(insert-ser-tab)
(begin-serialization (make-sys-path) (quote ("#" "cons-t")) (quote (s e c d i (j . "(car s)"))))
(new-ser-row (quote i) "(cdr s)")
(new-ser-row (quote i) "(car i)")
(new-ser-row (quote ((i . "(cons j i)") (j . "#:lt"))))
(new-ser-row (quote ((j . "(cdr s)") (s . "#:lt"))))
(new-ser-row (quote j) "(cdr j)")
(new-ser-row (quote ((s . "(cons i j)") (i . "#:lt") (j . "#:lt"))))
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote ((i . "#:lt") (c . "i"))))
(insert-ser-tab)
(begin-serialization (make-sys-path) (quote ("#" "eq-t")) (quote (s e c d i (j . "(car s)"))))
(new-ser-row (quote i) "(cdr s)")
(new-ser-row (quote i) "(car i)")
(new-ser-row (quote ((i . "(eq? j i)") (j . "#:lt"))))
(new-ser-row (quote ((j . "(cdr s)") (s . "#:lt"))))
(new-ser-row (quote j) "(cdr j)")
(new-ser-row (quote ((s . "(cons i j)") (i . "#:lt") (j . "#:lt"))))
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote ((i . "#:lt") (c . "i"))))
(insert-ser-tab)
(begin-serialization (make-sys-path) (quote ("#" "leq-t")) (quote (s e c d i (j . "(car s)"))))
(new-ser-row (quote i) "(cdr s)")
(new-ser-row (quote i) "(car i)")
(new-ser-row (quote ((i . "(<=? j i)") (j . "#:lt"))))
```

```
(new-ser-row (quote ((j . "(cdr s)") (s . "#:lt"))))
(new-ser-row (quote j) "(cdr j)")
(new-ser-row (quote ((s . "(cons i j)") (i . "#:lt") (j . "#:lt"))))
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote ((i . "#:lt") (c . "i"))))
(insert-ser-tab)
(begin-serialization (make-sys-path) (quote ("#" "add-t")) (quote (s e c d i (j . "(car s)"))))
(new-ser-row (quote i) "(cdr s)")
(new-ser-row (quote i) "(car i)")
(new-ser-row (quote ((i . "(plus j i)") (j . "#:lt"))))
(new-ser-row (quote ((j . "(cdr s)") (s . "#:lt"))))
(new-ser-row (quote j) "(cdr j)")
(new-ser-row (quote ((s . "(cons i j)") (i . "#:lt") (j . "#:lt"))))
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote ((i . "#:lt") (c . "i"))))
(insert-ser-tab)
(begin-serialization (make-sys-path) (quote ("#" "sub-t")) (quote (s e c d i (j . "(car s)"))))
(new-ser-row (quote i) "(cdr s)")
(new-ser-row (quote i) "(car i)")
(new-ser-row (quote ((i . "(minus j i)") (j . "#:lt"))))
(new-ser-row (quote ((j . "(cdr s)") (s . "#:lt"))))
(new-ser-row (quote j) "(cdr j)")
(new-ser-row (quote ((s . "(cons i j)") (i . "#:lt") (j . "#:lt"))))
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote ((i . "#:lt") (c . "i"))))
(insert-ser-tab)
(begin-serialization (make-sys-path) (quote ("#" "atom-t")) (quote (s e c d (i . "(car s)") j)))
(new-ser-row (quote i) "(atom? i)")
(new-ser-row (quote ((j . "(cdr s)") (s . "#:lt"))))
(new-ser-row (quote ((s . "(cons i j)") (i . "#:lt") (j . "#:lt"))))
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote ((i . "#:lt") (c . "i"))))
(insert-ser-tab)
(begin-serialization (make-sys-path) (quote ("#" "num-t")) (quote (s e c d (i . "(car s)") j)))
(new-ser-row (quote i) "(num? i)")
(new-ser-row (quote ((j . "(cdr s)") (s . "#:lt"))))
(new-ser-row (quote ((s . "(cons i j)") (i . "#:lt") (j . "#:lt"))))
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote ((i . "#:lt") (c . "i"))))
(insert-ser-tab)
(begin-serialization (make-sys-path) (quote ("#" "sym-t")) (quote (s e c d (i . "(car s)") j)))
(new-ser-row (quote i) "(sym? i)")
(new-ser-row (quote ((j . "(cdr s)") (s . "#:lt"))))
(new-ser-row (quote ((s . "(cons i j)") (i . "#:lt") (j . "#:lt"))))
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote ((i . "#:lt") (c . "i"))))
(insert-ser-tab)
(begin-serialization (make-sys-path) (quote ("#" "pair-t")) (quote (s e c d (i . "(car s)") j)))
(new-ser-row (quote i) "(pair? i)")
(new-ser-row (quote ((j . "(cdr s)") (s . "#:lt"))))
(new-ser-row (quote ((s . "(cons i j)") (i . "#:lt") (j . "#:lt"))))
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote ((i . "#:lt") (c . "i"))))
(insert-ser-tab)
(begin-serialization (make-sys-path) (quote ("#" "sl-t")) (quote (s e c d (i . "(car s)") j)))
(new-ser-row (quote i) "(sym-list i)")
(new-ser-row (quote ((j . "(cdr s)") (s . "#:lt"))))
(new-ser-row (quote ((s . "(cons i j)") (i . "#:lt") (j . "#:lt"))))
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote ((i . "#:lt") (c . "i"))))
(insert-ser-tab)
```

```
(begin-serialization (make-sys-path) (quote ("#" "ls-t")) (quote (s e c d (i . "(car s)") j)))
(new-ser-row (quote i) "(list-sym i)")
(new-ser-row (quote ((j . "(cdr s)") (s . "#:lt"))))
(new-ser-row (quote ((s . "(cons i j)") (i . "#:lt") (j . "#:lt"))))
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote ((i . "#:lt") (c . "i"))))
(insert-ser-tab)
(begin-serialization (make-sys-path) (quote ("#" "ci-t")) (quote (s e c d (i . "(car s)") j)))
(new-ser-row (quote i) "(char2int i)")
(new-ser-row (quote ((j . "(cdr s)") (s . "#:lt"))))
(new-ser-row (quote ((s . "(cons i j)") (i . "#:lt") (j . "#:lt"))))
(new-ser-row (quote ((c . "#:lt") (i . "(cdr c)"))))
(new-ser-row (quote ((i . "#:lt") (c . "i"))))
(insert-ser-tab)
(apply-data-refinement (make-sys-path) (quote ((s . s*) (e . e*) (c . c*) (d . d*) (i . i*) (j . j*)))
                                       (quote (mem)) (quote pr=>cell))
(resume-serialization (make-sys-path)
                      (quote ("rtn-t")) 1 2 (quote (s* e* c* d* i* (j* . "(alloc* mem)"))))
(new-ser-row (quote ((s* . "j*") (j* . "#"))))
(new-ser-row (quote j*) "(car* mem d*)")
(new-ser-row (quote ((mem . "(setcar!* mem s* i*)") (i* . "#"))))
(new-ser-row (quote ((mem . "(setcdr!* mem s* j*)") (j* . "#"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("dum-t")) 3 4
                                      (quote (s* e* c* d* i* (j* . "(alloc* mem)"))))
(new-ser-row (quote e*) "j*")
(new-ser-row (quote ((mem . "(setcdr!* mem j* i*)") (i* . "#") (j* . "#"))))
(ser-eval-ident (quote mem) (quote (setcdr!*-inits-car-to-nil)))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("ap-t")) 2 5
                                      (quote (s* e* c* d* (i* . "(alloc* mem)") j*)))
(new-ser-row (quote ((mem . "(setcdr!* mem i* j*)") (j* . "#"))))
(new-ser-row (quote j*) "(cdr* mem s*)")
(ser-eval-ident (quote j*) (quote (setcdr!*-effects-mem)))
(oblige-ser-eval-ident (quote j*) (make-term-path 0) "(lc-eq? (alloc* mem) s*)" "false")
(ser-eval-ident (quote j*) (make-term-path) (mk-sel-ident (quote false) "lc") (quote ()))
(new-ser-row (quote j*) "(car* mem j*)")
(ser-eval-ident (quote j*) (quote (car*-setcdr!*)))
(new-ser-row (quote ((mem . "(setcar!* mem i* j*)") (j* . "#"))))
(new-ser-row (quote ((e* . "#") (j* . "e*"))))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("ap-t")) 11 11
                                              (quote (s* e* c* d* i* (j* . "(alloc* mem)"))))
(new-ser-row (quote ((d* . "#") (mem . "(setcdr!* mem j* d*)"))))
(new-ser-row (quote ((i* . "#") (mem . "(setcar!* mem j* i*)"))))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("ap-t")) 14 14
                                      (quote (s* e* c* d* (i* . "(alloc* mem)") j*)))
(new-ser-row (quote ((j* . "#") (mem . "(setcdr!* mem i* j*)"))))
(new-ser-row (quote ((c* . "#") (mem . "(setcar!* mem i* c*)"))))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("ap-t")) 17 19
                                      (quote (s* e* c* d* i* (j* . "(alloc* mem)"))))
(new-ser-row (quote ((i* . "#") (mem . "(setcdr!* mem j* i*)"))))
(new-ser-row 'i* "(cdr* mem s*)")
(ser-eval-ident (quote i*) (quote (setcdr!*-effects-mem)))
(oblige-ser-eval-ident (quote i*) (make-term-path 0) "(lc-eq? (alloc* mem) s*)" "false")
(ser-eval-ident (quote i*) (make-term-path) (mk-sel-ident (quote false) "lc") (quote ()))
(new-ser-row 'i* "(cdr* mem i*)")
(ser-eval-ident (quote i*) (quote (setcdr!*-effects-mem)))
```

```
(ser-eval-ident (quote i*) (make-term-path 0) '(lc-eq?-false-2))
(ser-eval-ident (quote i*) (make-term-path) (mk-sel-ident (quote false) "lc") (quote ()))
(new-ser-row (quote ((i* . "#") (mem . "(setcar!* mem j* i*)"))))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("sel-t")) 8 8
                                      (quote (s* e* c* d* i* (j* . "(alloc* mem)"))))
(new-ser-row (quote ((i* . "#") (mem . "(setcar!* mem j* i*)"))))
(new-ser-row (quote ((d* . "#") (mem . "(setcdr!* mem j* d*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("ld-t")) 4 5
                                      (quote (s* e* c* d* i* (j* . "(alloc* mem)"))))
(new-ser-row (quote ((i* . "#") (mem . "(setcar!* mem j* i*)"))))
(new-ser-row (quote ((s* . "#") (i* . "s*"))))
(new-ser-row (quote ((s* . "j*"))))
(new-ser-row (quote ((i* . "#") (j* . "#") (mem . "(setcdr!* mem j* i*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("ldc-t")) 3 3
                                       (quote (s* e* c* d* i* (j* . "(alloc* mem)"))))
(new-ser-row (quote ((i* . "#") (mem . "(setcar!* mem j* i*)"))))
(new-ser-row (quote ((s* . "#") (mem . "(setcdr!* mem j* s*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("ldf-t")) 3 3
                                      (quote (s* e* c* d* i* (j* . "(alloc* mem)"))))
(new-ser-row (quote ((i* . "#") (mem . "(setcar!* mem j* i*)"))))
(new-ser-row (quote ((mem . "(setcdr!* mem j* e*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("ldf-t")) 6 6
                                      (quote (s* e* c* d* (i* . "(alloc* mem)") j*)))
(new-ser-row (quote ((j* . "#") (mem . "(setcar!* mem i* j*)"))))
(new-ser-row (quote ((s* . "#") (mem . "(setcdr!* mem i* s*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("exec-t")) 1 1
                                      (quote (s* e* c* d* i* (j* . "(alloc* mem)"))))
(new-ser-row (quote ((i* . "#") (mem . "(setcar!* mem j* i*)"))))
(new-ser-row (quote ((mem . "(setcdr!* mem j* e*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("exec-t")) 4 5
                                      (quote (s* e* c* d* (i* . "(alloc* mem)") j*)))
(new-ser-row (quote ((j* . "#") (mem . "(setcar!* mem i* j*)"))))
(new-ser-row (quote ((s* . "#") (j* . "(cdr* mem s*)"))))
(ser-eval-ident (quote j*) (quote (cdr*-setcar!*)))
(new-ser-row (quote ((j* . "#") (mem . "(setcdr!* mem i* j*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(new-ser-row (quote ((s* . "i*") (i* . "#"))))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("car-t")) 2 3
                                      (quote (s* e* c* d* i* (j* . "(alloc* mem)"))))
(new-ser-row (quote ((i* . "#") (mem . "(setcar!* mem j* i*)"))))
(new-ser-row (quote ((s* . "#") (i* . "(cdr* mem s*)"))))
(ser-eval-ident (quote i*) (quote (cdr*-setcar!*)))
(new-ser-row (quote ((i* . "#") (mem . "(setcdr!* mem j* i*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("cdr-t")) 2 3
                                      (quote (s* e* c* d* i* (j* . "(alloc* mem)"))))
```

```
(new-ser-row (quote ((i* . "#") (mem . "(setcar!* mem j* i*)"))))
(new-ser-row (quote ((s* . "#") (i* . "(cdr* mem s*)"))))
(ser-eval-ident (quote i*) (quote (cdr*-setcar!*)))
(new-ser-row (quote ((i* . "#") (mem . "(setcdr!* mem j* i*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("cons-t")) 1 6
                                      (quote (s* e* c* d* (i* . "(alloc* mem)") j*)))
(new-ser-row (quote ((j* . "#") (mem . "(setcar!* mem i* j*)"))))
(new-ser-row (quote ((j* . "(cdr* mem s*)"))))
(ser-eval-ident (quote j*) (quote (cdr*-setcar!*)))
(new-ser-row (quote ((j* . "(car* mem j*)"))))
(ser-eval-ident (quote j*) (quote (setcar!*-effects-mem)))
(ser-eval-ident (quote j*) (make-term-path 0) '(lc-eq?-false-2))
(ser-eval-ident (quote j*) (make-term-path) (mk-sel-ident (quote false) "lc") (quote ()))
(new-ser-row (quote ((j* . "#") (mem . "(setcdr!* mem i* j*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(new-ser-row 'j* "(alloc* mem)")
(new-ser-row (quote ((i* . "#") (mem . "(setcar!* mem j* i*)"))))
(new-ser-row (quote ((s* . "#") (i* . "(cdr* mem s*)"))))
(ser-eval-ident (quote i*) (quote (cdr*-setcar!*)))
(new-ser-row 'i* "(cdr* mem i*)")
(ser-eval-ident (quote i*) (quote (cdr*-setcar!*)))
(new-ser-row 's* "j*")
(new-ser-row (quote ((i* . "#") (j* . "#") (mem . "(setcdr!* mem j* i*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("eq-t")) 6 6 (quote ((s* . "i*") e* c* d* i* j*)))
(new-ser-row 'i* "(alloc* mem)")
(new-ser-row (quote ((s* . "#") (mem . "(setcar!* mem i* s*)"))))
(new-ser-row 's* "i*")
(new-ser-row (quote ((i* . "#") (j* . "#") (mem . "(setcdr!* mem i* j*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("leq-t")) 6 6 (quote ((s* . "i*") e* c* d* i* j*)))
(new-ser-row 'i* "(alloc* mem)")
(new-ser-row (quote ((s* . "#") (mem . "(setcar!* mem i* s*)"))))
(new-ser-row 's* "i*")
(new-ser-row (quote ((i* . "#") (j* . "#") (mem . "(setcdr!* mem i* j*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("add-t")) 5 6 (quote ((s* . "i*") e* c* d* i* j*)))
(new-ser-row 'j* "(cdr* mem j*)")
(new-ser-row 'i* "(alloc* mem)")
(new-ser-row (quote ((s* . "#") (mem . "(setcar!* mem i* s*)"))))
(new-ser-row 's* "i*")
(new-ser-row (quote ((i* . "#") (j* . "#") (mem . "(setcdr!* mem i* j*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("sub-t")) 5 6 (quote ((s* . "i*") e* c* d* i* j*)))
(new-ser-row 'j* "(cdr* mem j*)")
(new-ser-row 'i* "(alloc* mem)")
(new-ser-row (quote ((s* . "#") (mem . "(setcar!* mem i* s*)"))))
(new-ser-row 's* "i*")
(new-ser-row (quote ((i* . "#") (j* . "#") (mem . "(setcdr!* mem i* j*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("atom-t")) 2 3
                                      (quote (s* e* c* d* i* (j* . "(alloc* mem)"))))
(new-ser-row (quote ((i* . "#") (mem . "(setcar!* mem j* i*)"))))
(new-ser-row (quote ((s* . "#") (i* . "(cdr* mem s*)"))))
```

```
(ser-eval-ident (quote i*) (quote (cdr*-setcar!*)))
(new-ser-row (quote ((i* . "#") (mem . "(setcdr!* mem j* i*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(new-ser-row (quote ((s* . "j*") (j* . "#"))))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("num-t")) 2 3
                                      (quote (s* e* c* d* i* (j* . "(alloc* mem)"))))
(new-ser-row (quote ((i* . "#") (mem . "(setcar!* mem j* i*)"))))
(new-ser-row (quote ((s* . "#") (i* . "(cdr* mem s*)"))))
(ser-eval-ident (quote i*) (quote (cdr*-setcar!*)))
(new-ser-row (quote ((i* . "#") (mem . "(setcdr!* mem j* i*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(new-ser-row (quote ((s* . "j*") (j* . "#"))))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("sym-t")) 2 3
                                      (quote (s* e* c* d* i* (j* . "(alloc* mem)"))))
(new-ser-row (quote ((i* . "#") (mem . "(setcar!* mem j* i*)"))))
(new-ser-row (quote ((s* . "#") (i* . "(cdr* mem s*)"))))
(ser-eval-ident (quote i*) (quote (cdr*-setcar!*)))
(new-ser-row (quote ((i* . "#") (mem . "(setcdr!* mem j* i*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(new-ser-row (quote ((s* . "j*") (j* . "#"))))
(insert-ser-tab)
(resume-serialization (make-sys-path) (quote ("pair-t")) 2 3
                                      (quote (s* e* c* d* i* (j* . "(alloc* mem)"))))
(new-ser-row (quote ((i* . "#") (mem . "(setcar!* mem j* i*)"))))
(new-ser-row (quote ((s* . "#") (i* . "(cdr* mem s*)"))))
(ser-eval-ident (quote i*) (quote (cdr*-setcar!*)))
(new-ser-row (quote ((i* . "#") (mem . "(setcdr!* mem j* i*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(new-ser-row (quote ((s* . "j*") (j* . "#"))))
(insert-ser-tab)

(resume-serialization (make-sys-path) (quote ("sl-t")) 2 3
                                      (quote (s* e* c* d* i* (j* . "(alloc* mem)"))))
(new-ser-row (quote ((i* . "#") (mem . "(setcar!* mem j* i*)"))))
(new-ser-row (quote ((s* . "#") (i* . "(cdr* mem s*)"))))
(ser-eval-ident (quote i*) (quote (cdr*-setcar!*)))
(new-ser-row (quote ((i* . "#") (mem . "(setcdr!* mem j* i*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(new-ser-row (quote ((s* . "j*") (j* . "#"))))
(insert-ser-tab)

(resume-serialization (make-sys-path) (quote ("ls-t")) 2 3
                                      (quote (s* e* c* d* i* (j* . "(alloc* mem)"))))
(new-ser-row (quote ((i* . "#") (mem . "(setcar!* mem j* i*)"))))
(new-ser-row (quote ((s* . "#") (i* . "(cdr* mem s*)"))))
(ser-eval-ident (quote i*) (quote (cdr*-setcar!*)))
(new-ser-row (quote ((i* . "#") (mem . "(setcdr!* mem j* i*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(new-ser-row (quote ((s* . "j*") (j* . "#"))))
(insert-ser-tab)

(resume-serialization (make-sys-path) (quote ("ci-t")) 2 3
                                      (quote (s* e* c* d* i* (j* . "(alloc* mem)"))))
(new-ser-row (quote ((i* . "#") (mem . "(setcar!* mem j* i*)"))))
(new-ser-row (quote ((s* . "#") (i* . "(cdr* mem s*)"))))
(ser-eval-ident (quote i*) (quote (cdr*-setcar!*)))
(new-ser-row (quote ((i* . "#") (mem . "(setcdr!* mem j* i*)"))))
(ser-eval-ident (quote mem) (quote (setcar!*-setcdr!*-comm . rl)))
(new-ser-row (quote ((s* . "j*") (j* . "#"))))
```

```
(insert-ser-tab)

(apply-alg-ident (quote ()) (quote ("|2|" "sel-t")) "mem" (mk-sel-uni-br-ident "boolean" "lmem{lc}"))
(apply-alg-ident (quote ()) (quote ("|2|" "sel-t")) "j*" (quote assert2))
```