Data Refinement for Synchronous System Specification and Construction*

Alex Tsow and Steven D. Johnson

System Design Methods Laboratory,
Computer Science Department,
Indiana University

Abstract. *Design derivation*, a correct-by-construction system design method, specifies behavior with abstract datatypes. Refining these abstract datatypes is necessary for architectural decomposition. A new transformation primitive enables data refinement by generalizing term level injective homomorphisms to system equivalence.

Data refinement enables high levels of behavioral specification without sacrificing meaningful architectural decompositions. When behavior is specified over abstract datatypes the boundaries of a target architecture often cut across the borders of its datatypes. Consider SECD *(stack, environment, code, dump)*, an abstract machine for LISP's operational semantics, where each "register" holds nested pairs of atoms. Typical implementations represent this recursive datatype as reference(s) to a heap. The memory that holds the reference value (a register) and the memory that holds the heap cells (a RAM) are architecturally distinct. A target architecture which separates the register file from the RAM is exposed only upon refinement of the specification data types.

The standard underlying model is that of mutually corecursive equations defined over first-order terms with stream semantics. Below is a simple corecursive system of equations and its solution set (streams over integers).

```
X = 1 ! (+* X 1*)  X = ( 1,  2,  3, ...)
Y =     (-* X)     Y = ( -1, -2, -3, ...)
```
(1)

Definition of *sequential signals* is by destruction (expressed as arguments to the constructor `!`), i.e. `X` is a stream of integers whose head is the integer `1`, and whose tail is `(+* X 1*)`. 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 componentwise addition. The remaining signals are *combinational*, defined by lifted versions of term combinators; e.g. `Y` is the componentwise negation of `X`.

For simple lifting as just described, term level identities generalize to stream level identities, so local term *replacement* is one of the core transformations in the derivation algebra [1]. First order algebraic terms, unlike streams and other corecursive datatypes [3], are easily manipulated by standard theorem provers.

* This research is supported, in part, by the National Aeronautics and Space Association under the Graduate Student Researchers Program, NGT-1-010009.

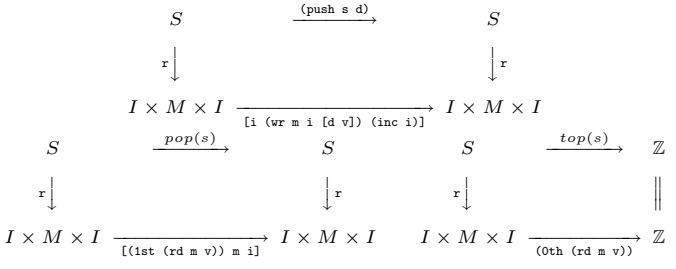
D. Borrione and W. Paul (Eds.): CHARME 2005, LNCS 3725, pp. 398–401, 2005.

Thus, local term replacement provides a hook for integration with other toolsets. By commutativity of addition, the equations

$$\texttt{X = 1 ! (+* X 1*)} \quad \text{and} \quad \texttt{X = 1 ! (+* 1* X)} \tag{2}$$

have the same solutions. Given this orthogonality, we usually eliminate the `*` annotation unless the context demands its use.

The refinement approach uses term level algebraic identities to express injective homomorphisms between types. The following diagram expresses the homomorphism `r` between abstract stacks of integers S and their implementation using references I to heaps expressed as a memory M of cells $I \times \mathbb{Z}$ addressed by I and a "next-unallocated-cell" pointer of type I.



(3)

`()` denotes function application, `[]` denotes tuple construction, integer ordinals are tuple accessors. The reference value, memory, and horizon pointer bindings are defined by `v=(0th (r s))`, `m=(1st (r s))`, and `i=(2nd (r s))`, respectively.

Even with a complete term-level characterization, term replacement is insufficient for local *representation translation*. Local translation allows different representations of the same abstract type in the same system, enables multi-level modeling, and promotes interaction. We can not change a sequential signal's type in the present system algebra. To this end, we introduce a new transformation which adds a new sequential signal of the implementation type, and replaces the target signal with an abstraction coercion from the implementation signal.

Theorem 1. *Let A and R be two sorts with functions* `r` $: A \to R$ *and* `a` $: R \to A$ *such that for all* `x` $\in R$, `(a (r x)) = x`. *Let* `X = x0 ! (T X)` *be an equation in a system description. Replacing* `X`*'s equation with*

```
X  =          (a X')
X' = (r x0) ! (r (T X))
```

(4)

preserves the solution for `X` *and*

```
(r X) = X'
```

(5)

is a valid stream-level identity.

This transformation combined with subsequent applications of the coercion identities over `r` (e.g. (3) and (5)), eliminates references to `X`'s in `X'`'s equation, thereby completing the refinement.

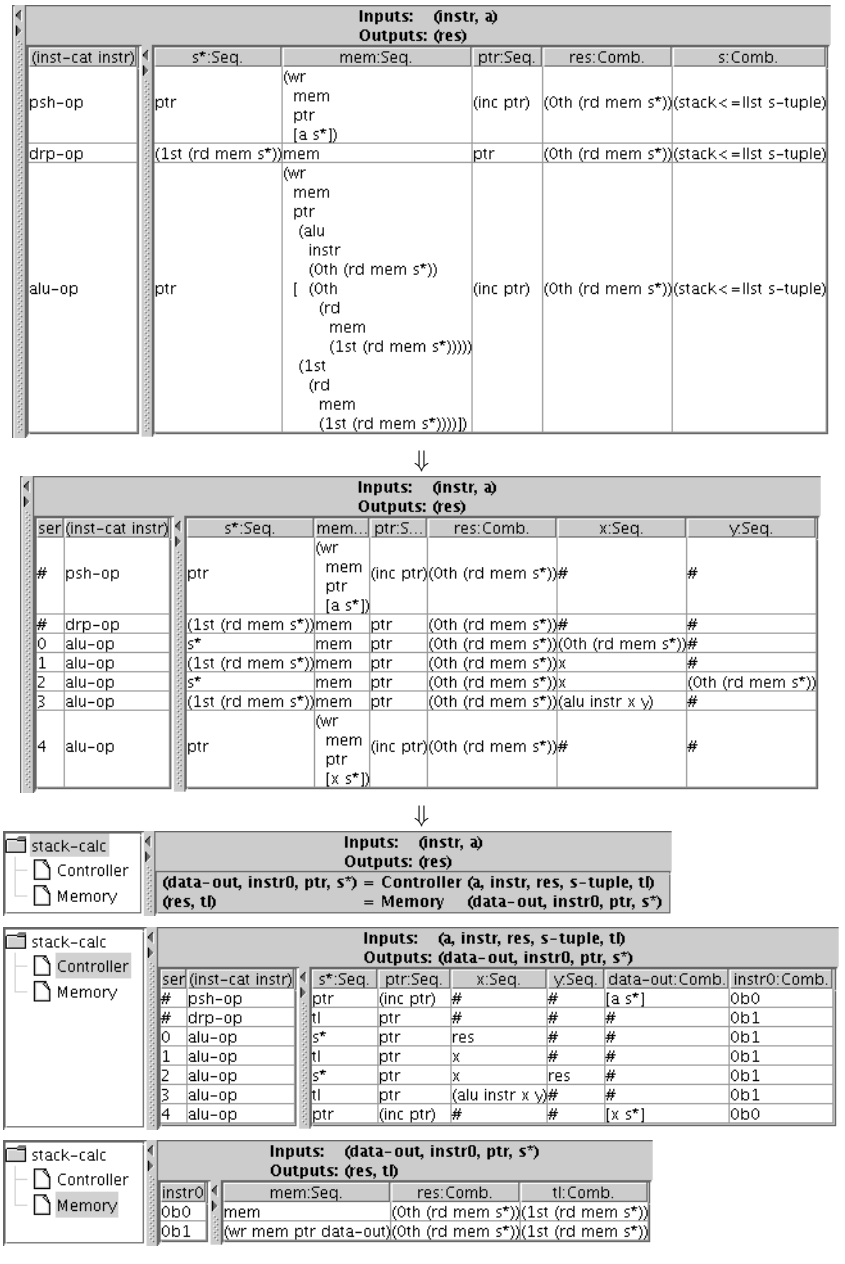
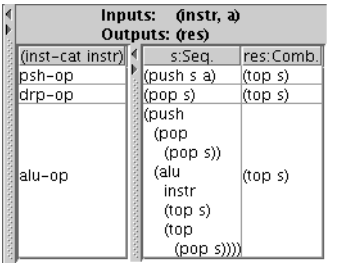


Fig. 1. The top table fully expand the refinement coercions, and splits the tupled signal into its three components as in (7). The next step *serializes* the deeply nested term guarded by `alu-op`. A column of integers in the decision table represent linear control flow that is invisible to external observers. The last step is a factorization that separates the memory from its referencing registers.

The following expressions are behavioral descriptions of a simple stack calculator. There are two input streams: an instruction token `instr` and an integer `a`. The function `inst-cat` classifies instructions as `psh-op`, `drp-op`, and `alu-op`.

The *behavior table* [2,4] explicitly enumerates the `case` key and branches in its decision table (left hand column), while indicating the signal updates in the action table (right-hand columns). Column headings indicate whether the signal is combinational or sequential. Current implementations omit display of initial values for sequential signals.

```
(stack-calc instr a) = res
 where
 s   = (push mt 0) !
       (case (inst-cat instr)
        (push s a)
        (pop s)
        (push
         (pop (pop s))
         (alu instr
          (top s)
          (top (pop s)))))
 res = (top s)
```



(6)

The application of Theorem 1 to sequential signal `s` in (6) generates a new signal `s'` of the implementation type, where `r` coerces the `case` statement. Functions commute with `case` branches; the first branch (the upper left cell of the action table) is rewritten using the identities from (3), (5), and explicit binding of components in `s'=[s* mem ptr]`:

```
(r (push s a))
 = [(2nd (r s)) (wr (1st (r s)) (2nd (r s)) [a (0th (r s))]) (inc (2nd (r s)))]
 = [(2nd s') (wr (1st s') (2nd s') [a (0th s')]) (inc (2nd s'))]
 = [ptr (wr mem ptr [a s*]) (inc ptr)]
```

(7)

This reduction continues in each `case` branch, corresponding to the behavior table rows for signal `s`. When complete, the tuple `s'=[s* mem ptr]` is split into its three component signals. Remaining references to the abstract type `s` are satisfied by the combinational application of the homomorphism's inverse: `s = (stack<=llst s')` (Starfish's coercion naming conventions are more verbose than `r` and `a`). Figure 1 shows the full expansion of refinement identities in (6), serialization of actions guarded by `alu-op`, and a factorization separating memory from its reference registers.

References

1. S. D. Johnson. Manipulating logical organization with system factorizations. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, July 1989.
2. S. D. Johnson and A. Tsow. Algebra of behavior tables. In *Lfm2000: Fifth NASA Langley Formal Methods Workshop, Proceedings*, 2000.
3. P. S. Miner. *Hardware Verification using Coinductive Assertions*. PhD thesis, Computer Science Department, Indiana University, USA, June 1998. T.R. No. 510.
4. A. Tsow and S. D. Johnson. Visualizing system factorizations with behavior tables. In *FMCAD 2000, Proceedings*, 2000.