Reconstructing Tempo in Haskell for C Program Specialization

Anonymous

Abstract A C specializer must preserve observable effects when only some inputs are known. We present a Haskell reconstruction of the model-directed interface in the historical Tempo distribution. The implementation propagates checked integer and immutable-table facts, specializes direct calls by argument value, and retains C execution for effects outside its abstract store. Two backends emit a C translation unit and a callable native library. We give a preservation argument for a sequential core, a resource bound on loop expansion and function versions, and a counterexample to unrestricted field substitution. Across thirteen published demo families, 32 distinct configurations produced 64 specialized builds that passed 1,224 checks. Two further specializations were generated from the complete SQLite 3.50.4 amalgamation. On each, 528 query rows and seven connection-limit transitions agreed with an independently compiled reference. The replication archive includes source snapshots, emitted code, native test paths, and per-case evidence.

1. Reconstruction target

Partial evaluation specializes a program with respect to information available before ordinary execution [1]. Tempo applied this principle to C, including pointer-bearing state, structured specialization scenarios, and both compile-time and runtime generation [2, 3]. This paper addresses a practical reconstruction question. Given a distribution containing a compiler image and examples, which behaviors can be independently implemented, exercised, and attributed to the new compiler?

The supplied archive contains 73 entries. Its primary compiler is an SML/NJ 110.0.7 heap image. Other entries include SUIF executables, OCaml front ends, x86 runtime templates, C support code, and models for power, dot product, and XDR. The main compiler's ML source is absent. We reconstructed the specialization workflow from the model grammar, the examples, and the public catalog. The resulting program has its own Haskell frontend, command line, and native generation path.

For a program generator operated by a person or an agent, equivalence on test outputs alone is insufficient evidence of specialization. A forwarding function can pass such a test. We record the residual argument list, inspect the generated body, and compare the compiled result with independent observations. The generated body shows where values were propagated and where the residual entry still calls generic code. The artifact includes the outputs on which this distinction is based.

2. Interface and operational contract

A model identifies a C translation unit, exported or internal procedures, external declarations, structure layouts, dependencies, and initialization expressions. A static annotation specifies a binding time. A concrete binding supplies a value. In the power model, an initializer in a separate C file returns the exponent four. In the dot-product model, each vector has a statically classifiable size, but specialization requires concrete sizes. Formal parameters are matched by name and residual arguments retain C declaration order. This detail matters because the supplied dot-product model and C definition list their two vectors in opposite orders.

The Haskell API separates preparation from individual requests. Preparation preprocesses and parses one complete translation unit and indexes its definitions. A prepared value supports multiple argument vectors. Each generated entry has a C prototype, residual body, original function bodies, and any specialized helpers. Keeping generic and residual definitions in one unit preserves access to file-scope names and permits calls back into original code. For deployment this unit replaces the corresponding input unit in the link.

Let S be the static argument space, D the remaining argument space, Σ the program state, and O^* the finite observable traces. For terminating deterministic executions of the core below, write

$$TX = \Sigma \rightharpoonup \left(X \times \Sigma \times O^*\right).$$

A C entry has denotation $f : S \times D \to TR$. Fixing s defines $j_s(d) = (s, d)$. Specialization must implement $f \circ j_s$. In the Kleisli category of the state-and-trace computation, this is composition with the pure embedding of j_s. If an entry binding specifies a field or global, restrict initial states to the predicate H. The equation then applies to every $\sigma \in H$. Returned values, observable writes, and ordered events are compared after projection away from private automatic storage. A bound field is an entry condition. Subsequent writes can change it.

3. Transformation rules

The scalar store Γ is a finite partial map from eligible C objects to known signed integers. Absence represents unknown information. Table facts require a compatible nonvolatile element type and an unambiguous initializer

layout. Each tracked pointer carries its current offset. Dynamic expressions remain C expressions. Integer evaluation uses Haskell integers for intermediate calculations, followed by a representability check for the target. Division and remainder use truncation toward zero. Undefined signed overflow, invalid shifts, and division by zero receive no computed constant. The generated unit checks the assumed integer widths. These choices follow the integer conversion and operator rules of the C11 committee draft [4].

For an ordinary assignment, the emitter preserves the C write and updates the fact about its destination. A final pass removes a restricted class of unused private integer stores. A residual call can mutate shared memory. The pass invalidates potentially affected entry facts before using the store again. Address-taken automatic integers are excluded from scalar propagation. In an expression with unspecified operand evaluation order, sibling effects invalidate facts used to simplify the other operand. Short-circuit evaluation only removes an operand on a path on which C would skip it.

If a condition is known, specialize its selected branch. For a dynamic condition with two continuing branches, the store at the join is

$$\operatorname{join}\ (\Gamma_1, \Gamma_2) = \{(k, v) \mid \Gamma_1(k) = v = \Gamma_2(k)\}.$$

A branch that returns contributes no state to the continuation. For example, the early null check in SQLite's column accessor permits the column index to remain known on the continuing path. This join rule follows the common-value interpretation of constant propagation [5].

For a loop with a known guard, the generator can emit one specialized body, update the store, and reconsider the guard. If the test becomes dynamic or an expansion budget expires, a residual C loop starts from the materialized current state. A fact about an object written by the residual loop is removed before its guard or body is simplified. Early exits and declarations with observable storage identity restrict copying. The generic path preserves the original control constructs.

For a direct call, the version key contains the target function and the known signed-integer arguments, including their formal names. A cache entry is installed before processing the callee. Recursive calls can therefore reuse a version under construction. Entry assumptions about mutable memory are excluded from scalar-only cache reuse. Function-local static storage, arbitrary jumps, and observable literal identity can require a wrapper that invokes the original function. Verbose output identifies these cases.

4. Two correctness obligations

Proposition 1. For a sequential fragment with nonescaping automatic signed integers, ordinary assignment, checked scalar expressions, conditionals, structured loops, and primitive effects with conservative write sets, the rules in Section 3 preserve each defined terminating execution satisfying its entry facts.

Argument. At each program point, require agreement between every retained fact and its concrete value. Constant substitution in a value position preserves value and trace by induction on expressions. The arithmetic checks preserve the C result and the short-circuit rule preserves evaluation order. An assignment either records the computed value or removes the fact. Invalidating the write set of a primitive effect reestablishes the invariant for every surviving entry. A dynamic join retains only facts true on every continuing path.

For an unrolled loop iteration, apply the statement induction hypothesis to the body and the invariant to the next guard. Induct on the number of emitted iterations. At a residual boundary every live object has backing storage, and removing the loop's possible writes makes the residual guard valid for all subsequent iterations. The dead-store rule applies only to unobserved private integer operations with an eligible right-hand side. These steps preserve the return value, visible final state, and finite trace. The argument is confined to the stated fragment. The implementation's handling of full GNU C, the parser, the pretty printer, the host compiler, dynamic linking, and ABI conversions require independent evidence.

Proposition 2. With helper limit V and per-function iteration budget F, at most $1 + V$ new function definitions are emitted for one entry and at most $(1 + V)F$ successful loop-iteration expansions occur.

Argument. A fresh helper consumes one slot before descending into its body. Reuse consumes no slot. Every successful loop expansion decrements a counter shared by all loops in the function. At zero, code generation retains a call or a loop. The count bounds expansions. The source size of one copied body and the work required to parse the original translation unit are separate costs.

Aliasing counterexample. Suppose a initially points to a row of size 5. Read its size into n, write -7 to the size of a row through b, then return `n + a->size`. With distinct rows the result is 10. With $a = b$ it is -2. Replacing both reads by the entry value would give 10 in the aliasing case. Tests exercise both direct pointer writes and mutation through a call. The observed store and return agree with the original on both alias arrangements.

5. Historical program evidence

For the supplied power example, the residual entry accepts only the base. Its body contains four multiplications and no exponent counter. With a fuel budget of two and an exponent of nine, a residual loop finishes the calculation. For dot product with both sizes equal to three, the residual body contains indexed products at 0, 1, and 2. In the supplied XDR encoder, a specialized construction helper writes operation 0 and span 4. Subsequent serialization and external byte-order conversion remain executable C.

The public catalog contains thirteen demo families [6]. We extracted the documented static configurations, compiled the generic C independently, and tested both the C-emission path and the native library path. Counts in the table are **per path**. A check is a harness assertion. Its granularity varies by demo.

Demo group	Configurations	Checks
Context, flow, return	3	44
Three interpreters	3	31
Power and formatted power	7	35
RPC and packet filter	2	29
Splines and matrix product	6	421
Romberg, Chebyshev, FFT	11	52
Total per path	**32**	**612**

In the context example, $g(3, d)$, $g(7, d)$, $g(d, 3)$, and $g(d, 7)$ produce four helper versions. The combined result is $4d + 32$. In the return example the specialized helper still writes the global $x = 1 + d$ and returns the constant 3. The caller uses the result and produces $d + 5$. The emitted helper body supplies evidence for the intermediate write, which the final value alone cannot distinguish.

The first interpreter reduces to straight-line operations over the original stack and memory. The other two programs compute factorial using dynamic control, with a loop in one case and recursion in the other. The tests compare all memory cells, the complete stack, and emitted output for each initial state. For the numerical cases the independent observations include the spline tridiagonal equations, a polynomial integral, cosine-projection coefficients, and a direct complex DFT. The FFT lengths are 16, 32, 64, and 128. The largest absolute complex-error tolerance is 0.003 for the single-precision FFT. Matrix cases include all four supplied vector files and check each computed entry against an independent dot product.

The published RPC example is a reduced SunOS client. Its documentation records the removal of reply decoding and the missing execution support [7]. The LP64 adapter restores initialization and required operations, serializes four-byte network-order words, and supplies a localhost UDP receiver. A test compares all 44 request bytes, including the header, procedure number, and five signed integers. Other tests exercise an undersized XDR buffer and a failed send. The filter demo uses the historical capture and four boundary inputs. Its fixed program specializes to an initial length check, packet-word load, and conditional calls. The original filter evaluates the remaining instruction suffixes. These results concern the ported client and the offline capture.

6. SQLite as a complete-source test

SQLite 3.50.4 was pinned by the published SHA3-256 digest of its amalgamation [8]. The uncompressed source contains 9,282,866 bytes and 262,899 lines. Preprocessing with the recorded GCC options exposes 2,574 function definitions. The parser input normalizes a standalone GNU fall-through diagnostic attribute to an empty statement. The emitted original translation unit retains those attributes. The specializer receives the complete C file for each experiment.

For the first experiment, fix column index zero in the public integer-result accessor. The generated function calls a specialized internal column lookup containing the expression `pResultRow[0]`. It retains the bounds check, mutex operations, conversion and error handling. The tests include grouped results, joins, negative integers, NULL, text, BLOB, and floating-to-integer conversions. Results were compared on independent in-memory databases. SQLite numbers its leftmost result column zero [9].

For the second experiment, fix the limit category to the SQL text-length limit and leave its new value dynamic. The residual source accesses `aLimit[1]`. Tests check the previous value returned, the resulting connection state, a query-only call, acceptance of a short statement, and rejection of an overlength statement. This exercises a state-changing API with a different observational contract from column retrieval [10]. Both independently compiled specialized units reproduced the same 528 two-column rows and seven limit transitions as the stock library. The scope of the experiment is complete-source processing with behavioral validation at these two entry points.

7. Measurement and replication

Five fresh-process rounds measured complete generation on the SQLite source. The experiment rotated the order of standalone C preprocessing, column specialization, and limit specialization. Timing includes process startup and output-file creation. Native compilation of

SQLite and SQL execution are outside the measured interval. Compression and hashing occurred outside the timer. The reference and specialized libraries in the behavioral comparison used the same C optimization level.

Stage	Median (s)	Range (s)	Peak (MiB)
Preprocess	0.152	0.150–0.225	49.8
Column	1.620	1.471–2.263	307.0
Limit	1.530	1.479–1.914	307.0

Peak memory is the median of the five Linux maximum resident-set measurements in each stage. The host reported an AMD EPYC 9V74 processor and GCC 14.2.0. The generated definitions for the column and limit cases occupied 1,112 and 768 bytes of C source respectively, in addition to the retained translation unit.

The timing data characterize this build and machine. For repeated specialization of one function, the library interface can reuse a prepared input. Application throughput and the amortization point for native compilation require workload-specific measurements. The complete generation CSV, option list, processor model, compiler version, source digest, and emitted C are included in the archive.

The regression suite passed 1,405 checks. A Haskell caller made 21 native calls over three specializations of one prepared unit. The replication package contains a Cabal project, a prebuilt Linux executable, upstream input hashes, portability modifications, the original example models, the SQLite fixture, and both source and native harnesses. The historical comparison uses the generic C programs and independent observables. Compatibility covers the documented scenarios exercised here. The historical heap analysis, GUI, and x86 template ABI fall outside this reconstruction. An implementation error shared by the generic compiler and the residual compiler can survive differential testing. Independent mathematical calculations and the byte-level wire oracle reduce that common-mode risk. Compiler differential testing has previously exposed wrong-code defects in production toolchains [11].

The public catalog can be rerun with `python3 demos/run.py`. The SQLite comparison runs with `python3 test/sqlite.py`. The archive and language regressions run with `python3 test/integration.py`. The native Haskell caller is exercised by `cabal test native-api`. The Haskell source, all specialization contexts, and the raw evidence identify which behaviors can be reproduced independently of the old binaries.

Contributor disclosure. Anonymous review copy. A human requester specified the reconstruction target and evaluation scope. OpenAI Codex generated the implementation, ran the experiments, and drafted the manuscript.

References

1. N. D. Jones, C. K. Gomard and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993. Book
2. C. Consel, J. L. Lawall and A.-F. Le Meur. “A tour of Tempo: a program specializer for the C language.” *Science of Computer Programming* 52, 341–370, 2004. Publisher
3. IRISA / INRIA. *Tempo Specializer.* Project overview, analysis and generation phases. Project documentation
4. ISO/IEC JTC1/SC22/WG14. *Programming Languages — C.* Committee draft N1570, 2011. §§6.2.4, 6.3.1, 6.5 and 6.8. Draft
5. M. N. Wegman and F. K. Zadeck. “Constant propagation with conditional branches.” *ACM Transactions on Programming Languages and Systems* 13(2), 181–210, 1991. IBM Research
6. IRISA / INRIA. *Tempo Specializer — Demos.* Version 1.194, 1998. Catalog and source directories
7. IRISA / INRIA. *Reduced Sun RPC Client Example.* Upstream implementation notes
8. SQLite. *Release 3.50.4*, July 30, 2025. Release and source digest
9. SQLite. *Result Values from a Query.* C interface
10. SQLite. *Run-time Limits.* C interface
11. X. Yang, Y. Chen, E. Eide and J. Regehr. “Finding and Understanding Bugs in C Compilers.” *PLDI*, 2011. University of Utah