A Certified Framework for Compiling and Executing Garbage-collected Languages

Andrew McCreight Tim Chevalier Andrew Tolmach
Portland State University
{mccreigh,tjc,apt}@cs.pdx.edu

Abstract

We describe the design, implementation, and use of a machine-certified framework for correct compilation and execution of programs in garbage-collected languages. Our framework extends Leroy's Coq-certified Compcert compiler and Cminor intermediate language. We add: (i) a new intermediate language, GCminor, that includes primitives for allocating memory in a garbage-collected heap and for specifying GC roots; (ii) a precise, low-level specification for a Cminor library for garbage collection; and (iii) a proven semantics-preserving translation from GCminor to Cminor plus the GC library. GCminor neatly encapsulates the interface between mutator and collector code, while remaining simple and flexible enough to be used with a wide variety of source languages and collector styles. Front ends targeting GCminor can be implemented using any compiler technology and any desired degree of verification, including full semantics preservation, type preservation, or informal trust.

As an example application of our framework, we describe a compiler for Haskell that translates the Glasgow Haskell Compiler's Core intermediate language to GCminor. To support a simple but useful memory safety argument for this compiler, the front end uses a novel combination of type preservation and runtime checks, which is of independent interest.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers, Memory management (garbage collection); D.2.4 [*Software/Program Verification*]: Correctness proofs

General Terms Languages, Verification, Reliability, Security.

1. Introduction

Programming in high-level, type-safe languages such as Haskell, ML, or Java eliminates large classes of potential bugs, thus increasing reliability while reducing implementation time and cost in many application domains. Safe languages should be particularly attractive for implementing systems that demand the highest possible levels of assurance, such as safety-critical device control or high-security data processing, which are currently very expensive to produce. But the appeal of these languages for high-assurance applications is undercut by their reliance on large, complex runtime systems, usually written in C or assembler. For example, the runtime system of the Glasgow Haskell Compiler (GHC) [31] consists of roughly 75,000 lines of C code. Such systems are very difficult to verify, even informally.

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.

Garbage collection (GC) is a key runtime service that is often a source of bugs. GC bugs can result from erroneous algorithms or incorrect collector implementations, or because the intended interface between the collector and the *mutator*—the application code that makes allocation requests and performs reads and writes on the heap—has been violated. Moreover, GC bugs are often difficult to reproduce and diagnose. Garbage collection is therefore a good application area for formal methods, including machine-certified correctness proofs, and several proofs of collector implementations have been developed in recent years [14, 22, 24, 36].

Bugs often occur because the collector-mutator interface has not been explicitly specified, making it easy for implementers on either side to violate intended invariants. Precise garbage collectors must be able to access all *roots*, i.e., pointers from mutator data structures into the heap, and to ascertain the *layout*, i.e., size and embedded pointer positions, for all heap records. The collector proofs cited above formalize the interface as seen from the collector's side. But there has been much less work on ensuring that the mutator obeys its side of the interface.

In this work, we show how to encapsulate the key aspects of the collector-mutator interface into the semantics of a generic *intermediate language*, called GCminor, that can serve as the target for compiling a range of garbage-collected languages, and as the source for a machine-certified semantics-preserving compiler back end. GCminor makes the collector-mutator interface both explicit and precise. A *client*, i.e., a compiler front end, can use the back end simply by generating code in GCminor and record layout descriptions in a format that GCminor can accept.

GCminor supports many styles of uniprocessor memory managers, including mark-sweep, copying, and generational collectors. Any real collector implementation will modify the heap and possibly the values of root variables. However, GCminor's semantics completely hide these effects: from the perspective of mutator code, the heap and reachable pointers do not appear to change at all during a collection. This property makes it much easier to verify that the mutator code obeys the GC interface. We enforce correctness of root declarations using a block-based memory model together with a novel pointer-clearing technique at the semantic level.

Our work extends Leroy's Compcert compiler and Cminor intermediate language [16, 18]. Compcert compiles (most of) C to PowerPC or ARM assembly code, and is proven to preserve the observable behavior of the program: its sequence of system calls and final return value. The proof is certified using the Coq Proof Assistant [2]. Cminor is an untyped, low-level imperative intermediate language with C-like control constructs, which sits between Compcert's C-specific front end and its processor-specific back end. It supports global and local memory, but not a heap.

We define GCminor as a small extension to Cminor that adds statements for allocating in a garbage-collected heap and declarations for specifying heap roots. We implement new Compcert pipeline stages to translate GCminor programs into ordinary Cminor code, intended to be linked against a memory management library also written in Cminor. The existing Compcert back end com-

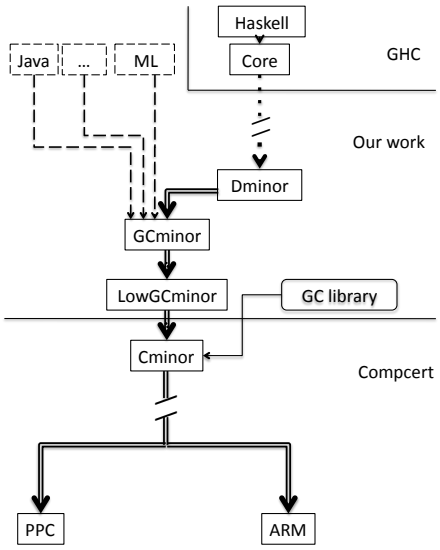


Figure 1. Overall architecture. Boxes represent languages; double lines are semantics-preserving translations and dotted lines are type-preserving translations. The GC library, written in Cminor, is linked into the program during translation. The dashed boxes and lines show possible future extensions.

piles the resulting complete Cminor program to machine code, and the existing back end semantics-preservation proofs guarantee the executable's runtime behavior. The translation from GCminor to Cminor is proven in Coq to be semantics-preserving relative to a low-level specification of the memory management library, which can be implemented by a range of bump-pointer allocators and record-moving collectors. In particular, several existing proofs of Cheney-style copying collectors, developed both by ourselves [22] and others [14, 24], obey similar specifications. (We do not yet have a machine-checked proof that the specification assumed by GCminor and the specification obeyed by our collector match precisely.)

To illustrate the utility of our extended Compcert back end, we exhibit a prototype Haskell compiler. We implement a front end that translates GHC's Core intermediate language to another new intermediate language, dubbed Dminor, and from there to GCminor. Dminor is a purely functional, typed language that guarantees memory safety through a novel combination of runtime checks and a rudimentary type system to distinguish pointers from non-pointers. We have proven semantics preservation in Coq for most of the Dminor-to-GCminor translation (including all the language features that involve allocation). This result can be combined with a type checker for Dminor and soundness proof for the checker to show that any program that compiles without complaint is memory-safe. To obtain a stronger guarantee, we could prove that the Core-to-Dminor translation preserves well-typedness, thereby showing that any well-typed Core program is memory-safe. (We do not have machine-checked versions of these typing proofs.)

2. Compcert and Cminor

Our work forms a backwards-compatible extension to Version 1.4 of the Compcert system [16–18], which we review in this section. Figure 1 shows the overall architecture of our framework.

Compcert and its correctness proof are structured as a pipeline of small translation steps between intermediate languages, each with its own operational semantics. The compiler itself is a purely

$$
\begin{array}{lll}
e ::= & id & \text{local variable} \\
 & \mid l & \text{integer or float literal} \\
 & \mid \texttt{addrsymbol(}id\texttt{)} & \text{address of global symbol} \\
 & \mid \texttt{stackaddr}_i & \text{address of stack frame entry} \\
 & \mid op(\vec{e}) & \text{arithmetic operations} \\
 & \mid \texttt{load(}ch\texttt{,}e_{addr}\texttt{)} & \text{memory load} \\
 & \mid e_1 \texttt{ ? } e_2 \texttt{ : } e_3 & \text{conditional expression} \\
 & & \\
s ::= & id = e\texttt{;} & \text{local variable assignment} \\
 & \mid \texttt{store(}ch\texttt{,}e_{addr}\texttt{,}e_{val}\texttt{);} & \text{store to global or stack frame} \\
 & \mid \texttt{[}id = \texttt{]}\ \texttt{call(}e\texttt{,}\vec{e}\texttt{);} & \text{function call} \\
 & \mid \texttt{tailcall(}e\texttt{,}\vec{e}\texttt{);} & \text{function tail call} \\
 & \mid \texttt{if } e\ \{\vec{s}_1\}\ \texttt{else}\ \{\vec{s}_2\} & \text{conditional} \\
 & \mid \texttt{block}\ \{\vec{s}\} & \text{delimited block} \\
 & \mid \texttt{loop}\ \{\vec{s}\} & \text{infinite loop} \\
 & \mid \texttt{exit}\ n\texttt{;} & \text{block exit} \\
 & \mid \texttt{switch}\ e\ \{\overrightarrow{i : n}\}\ n\texttt{;} & \text{switched exit} \\
 & \mid \texttt{return}\ \texttt{[}e\texttt{]}\texttt{;} & \text{return} \\
 & \mid \texttt{skip;} & \text{no-op}
\end{array}
$$

$$f ::= \texttt{fun(}\overrightarrow{id}\texttt{)}\ \{\ \texttt{stack}\ n\texttt{;}\ \texttt{vars}\ \overrightarrow{id}\texttt{;}\ \vec{s}\ \}$$

$$
\begin{array}{ll}
p ::= & \texttt{functions}\ \overrightarrow{id = f}\texttt{;} \\
 & \texttt{vars}\ \overrightarrow{id = initializers}\texttt{;} \\
 & \texttt{main}\ id
\end{array}
$$

Figure 2. Syntax of Cminor expressions (e), statements (s), functions (f) and programs (p). A chunk (ch) specifies type, size, and signedness of a datum.

functional Coq program that is automatically extracted to executable OCaml code [34].

Cminor. Cminor (Figure 2) is the C-like intermediate language at the heart of the Compcert compiler. A program consists of function definitions and global data definitions. Functions have the usual parameters and local variables. In addition, a function can place data in an explicit stack frame, which is disjoint from storage for the named variables. The size of a function's frame is specified in the function header; the expression $\texttt{stackaddr}_i$ refers to the ith byte of the current function's frame. Other expressions are standard. Statements are also largely standard, but include support for structured control flow using nested `blocks`; an `exit` statement, where `exit` n branches to the end of the $(n+1)$st enclosing block; and a `switch` statement, which matches an integer discriminant against a list of values and exits to the corresponding specified enclosing block.

Cminor semantics. Cminor has a small-step operational semantics in the style originally suggested by Appel and Blazy [3]. A characteristic transition rule for Cminor statements has the form

$$G \vdash_{CM} (F, st, E, k, \sigma, M) \xrightarrow{t} (F', st', E', k', \sigma', M')$$

Here G is the global environment, which maps function names to definitions and global variables to memory locations; F is the current function definition; st is the statement at the current program point; E is the local environment, which maps parameters and locals to values; k is the continuation, which describes both the remainder of the current function and the call stack, including the local environments of suspended activations; σ points to the explicit stack frame; M is the memory; and t is a trace of the observable events (system calls) that occur as a side-effect of evaluation.

The meaning of a program is given by the (finite or infinite) trace it produces when started from a suitable initial state, together with its final result value (if it terminates). As usual, possible

unchecked errors during program execution are modeled by the absence of a suitable transition rule, in which case evaluation is said to "get stuck."

A distinctive feature of the Compcert architecture is that all intermediate languages in the pipeline, from C through assembly code, share the same memory model [19] and notion of values. The memory M is composed of an unbounded number of disjoint blocks, indexed by (mathematical) integers. Cminor uses one statically-allocated block per global variable and one dynamically-allocated block per stack frame. Each block has fixed upper and lower bounds (signed mathematical integers) set at block-allocation time. The block contains values indexed by byte offsets (signed machine integers) within these bounds. Operations on the memory include *Mem.alloc*, which (always) returns a fresh block, with initially undefined contents; *Mem.high_bound* and *Mem.low_bound*, which return the bounds of a specified block; *Mem.load* and *Mem.store*, which operate on a specified block, offset, and *chunk*; and *Mem.free*, which renders a block inaccessible (but does not permit the block number to be re-used). Chunks describe the type, size, and signedness of memory being accessed; they include *int8unsigned*, *int32* and *float64*.

Values are described by the grammar

$$\mathit{value} := \mathit{Vint}\ n \mid \mathit{Vfloat}\ f \mid \mathit{Vptr}\ b\ n \mid \mathit{Vundef}$$

where n is a 32-bit machine integer (which can represent both signed and unsigned numbers), f is a double-precision IEEE float, and $b \in \mathbb{Z}$ is a memory block number. *Vptr b n* is a pointer to offset n within block b. The null pointer is represented by *Vint 0*. *Vundef* represents undefined values, e.g., the contents of uninitialized offsets within a block.

Memory loads and stores only succeed within the boundaries of a valid block. Also, it is not possible to cast one kind of value to another (without applying an explicit coercion operator), so in particular, *Vptr* values cannot be forged.

As a concrete example of a semantics transition rule, here is one for the `store` statement:

$$\frac{\begin{array}{c} G, E, M, \sigma \vdash e_a \rightarrow \mathit{Vptr}\ b\ n \\ G, E, M, \sigma \vdash e_v \rightarrow v \\ \mathit{Mem.store}\ M\ ch\ b\ n\ v = \mathit{Some}\ M' \end{array}}{G \vdash_{CM} (F, \texttt{store(}ch\texttt{,}e_a\texttt{,}e_v\texttt{)}, E, k, \sigma, M) \xrightarrow{\epsilon} (F, \texttt{skip}, E, k, \sigma, M')}$$

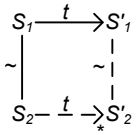
Here the hypotheses of the form $G, E, M, \sigma \vdash e \rightarrow v$ invoke a separate set of rules for evaluating expressions e to values v. If the parameters to *Mem.store* are invalid, the rule will not apply, and the program will get stuck. For statements such as `store` that do not alter control flow, we define the next statement to be just `skip`; we can then encapsulate all the details of inspecting the continuation to determine what to do next within the transition rule for `skip`. The trace annotation ϵ represents the empty trace.

Assembly code semantics. Other intermediate languages in the Compcert pipeline use similar small-step semantic formulations. For example, the assembly code semantics transition relation has the form

$$G \vdash_{AS} (R, M) \xrightarrow{t} (R', M')$$

where R maps target machine registers to values, and G, M, and t are as above. As with Cminor, the assembly code semantics uses one statically-allocated block per global variable and one dynamically-allocated block per stack frame. Note that because this semantics uses the same models of values and memory as earlier languages in the pipeline, assembly code programs will get stuck if they attempt to forge pointers or access arbitrary parts of memory. Thus, "non-stuck" assembly programs enjoy a memory safety property. This property is somewhat accidental, in the sense that if Compcert elected to use a lower-level memory model for assembly code—e.g., a flat array of bytes—then progress of assembly programs might not imply anything about memory safety.[1]

Semantics preservation. Semantics preservation proofs in Compcert generally take the form of forward simulations. Let L_1 and L_2 be adjacent languages in the compiler pipeline, P_1 be a program in L_1 and P_2 be the corresponding program in L_2. To show that P_1 and P_2 have the same observable behavior, we define a simulation relation $\sim$ between the semantic states of L_1 and those of L_2, and then show that this relation is preserved as execution of P_1 and P_2 progresses.



In words, the diagram says that if state S_1 is simulated by state S_2, and S_1 can reach S_1' by taking a single step, generating trace t, then there exists a state S_2' such that S_2 can reach S_2' by taking zero or more steps also generating t, and S_1' is simulated by S_2'. By inductively applying this diagram over an entire L_1 execution sequence, we can prove the existence of an equivalent L_2 sequence: that is, any observable behavior of P_1 can be mimicked by P_2. Moreover, if P_2 is deterministic, we can also show that every P_2 behavior is equivalent to a P_1 behavior, and hence that the translation preserves specifications about the behavior; see Leroy [18] for details.

Memory embeddings. In many cases, a key part of the state simulation relation $S_1 \sim S_2$ describes how the memory components M_1 and M_2 of the two states are related. Depending on the transformation, memory blocks may be added, removed, or combined. Formally, the memory relation is specified by an *memory embedding* ϕ from the blocks of M_1 to addresses in M_2 [19]. For a semantics preservation proof to succeed, the embedding must guarantee that successful loads and stores in M_1 are simulated in M_2.

3. GCminor

GCminor is our target language for generated mutator code. A primary design goal for GCminor is that it be as general-purpose as possible. On the mutator side, we support both functional and object-oriented languages. The principal restriction on clients is that heap roots must be identified statically; we do not support collectors that distinguish dynamically between pointer and non-pointer values, e.g., by stealing a bit from each value to flag pointers, a trick that cannot be expressed in Compcert's current memory model. This limitation makes GCminor unsuitable as a target for compilers that generate a single piece of object code for functions that are polymorphic over both boxed and unboxed values. On the collector side we support a range of "stop-and-collect" styles, including both copying and mark-sweep collectors; we also include hooks supporting generational collectors. We leave extension to incremental and concurrent collectors for future work.

GCminor provides a well-defined interface for communication between the mutator and collector. There are two main aspects to the interface:

i. Garbage collection *roots* are mutator variables that hold pointers to heap records. The collector uses these as the starting

[1] On the other hand, informally the existence of the observational equivalence proof would still be very comforting, as it is hard to imagine that a (non-malicious) compiler could systematically violate memory safety and still generate correct code for all programs!

s ::= ... as in Cminor
| $\overrightarrow{id}$ `=` `alloc(`$\vec{i}$`);` heap record allocation
| `rstore(`ch`,`e_{bl}`,`e_{off}`,`e_{val}`);` store to heap record
| *[id =]* `extcall(`e`,`$\vec{e}$`);` call to external function

f ::= `fun(`$\overrightarrow{id}$`) { stack` n`; vars` $\overrightarrow{id}$`; roots` $\overrightarrow{id}$`;` $\vec{s}$ `}`

Figure 3. Syntax of GCminor statements(s) and functions (f). Expressions (e) and programs (p) are the same as in Cminor.

points of its search for reachable records; if a moving collector is used, it may also update root values. Roots are explicitly declared in GCminor functions.

ii. The *layout* for a heap record tells the collector how long the record is, and which fields contain pointers. As with roots, pointer fields must be traced to find other reachable records, and may also be updated by a moving collector. In our system, pointer layout is always determined by the record header, but the precise method by which this is done is an auxiliary parameter of the system, specified outside of GCminor itself.

Language. GCminor (see Figure 3) extends Cminor with three syntactic features that support a garbage-collected heap. First, the language is augmented with an `alloc` statement that allocates fresh heap records of specified sizes, after performing garbage collection if necessary. Second, each function definition is annotated with a list of the variables (parameters and locals) that hold heap pointers; the garbage collector uses these variables as roots for its traversal of the live data graph within the heap. Finally, `rstore` is a new variant of the memory `store` statement specifically for updating heap records, which provides a hook for a write barrier. We also add a syntactic distinction between calls to GCminor code and calls to external functions, which lets the implementation of GCminor use different calling conventions for these two cases (see Section 4).

The statement id_1`,...,`id_n `= alloc(`i_1`,...,`i_n`)` allocates n records with data sizes $i_1, \ldots, i_n$ (counted in words) and assigns pointers to them into $id_1, \ldots, id_n$ respectively. Each record is prefixed by an additional one-word header, which is not included in the size argument. The `alloc` statement does *not* initialize the records; it is the responsibility of the mutator to keep the heap *well-formed* by filling in the header and data fields consistently before the next allocation, as described in more detail below. If there is insufficient heap memory for the requested records even after a collection, the program's behavior is undefined (semantically, it gets stuck); in practice, the allocator issues a runtime error.

The `alloc` statement supports multiple simultaneous allocations; this is necessary to allow the mutator to build mutually recursive records efficiently. If the GCminor implementation allocates by bumping a free pointer (as with a copying collector), it can combine the storage requirements of the multi-allocation before checking the heap limit. Thus, mutators can make natural use of multi-allocation to obtain code that does just one limit check per basic block, an important common optimization. At the same time, keeping the allocations distinct at the GCminor level preserves the possibility of switching transparently to an underlying allocator that uses free lists (as in a mark-sweep collector).

Any allocation may invoke the garbage collector, which calculates the set of live heap records and reclaims the space used by dead (garbage) records for use in subsequent allocations. More precisely, the collector computes the set of records that are *reachable* via a chain of heap pointer dereferences starting from the roots declared in the current function activation and all suspended activations. Roots are either local variables or function parameters. For simplicity, GCminor does not currently support roots in global data regions or in explicitly-allocated stack frames; these restrictions would be straightforward to remove. Correct specification of roots by the mutator is essential, because only pointers to reachable records are guaranteed to remain valid after a collection. Mutator code must obey the invariant that each declared root variable always holds either a heap record pointer obtained from an `alloc` or `null`. To make this task easier, GCminor implicitly initializes all local roots to `null`; the subsequent translation from GCminor to Cminor can usually eliminate these initializations.

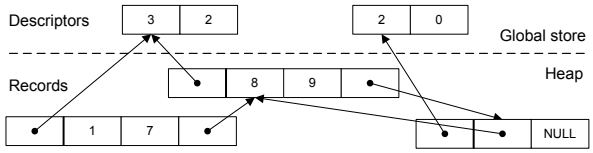


Figure 4. Example of standard layout descriptor scheme

Record layout. Both the semantics of GCminor and the actual implementation of the underlying collector need to know which record fields contain heap pointers. We classify all values as having either GC type *Ptr*, meaning a heap pointer, or GC type *Atomic*, meaning an integer, float,[2] or a pointer into the global static data area. (The value *Vint 0*, which represents `null`, has both types.) The collector needs to identify all fields that contain *Ptr* values.

Concretely, there are many possible ways to associate a record's header with its layout description: e.g., the size and pointer information might be stored directly in bit fields within the header, or indirectly in an auxiliary data structure pointed to by the header. When designing GCminor, we considered hiding this choice from clients, and, e.g., simply including a list of pointer field locations as an additional parameter of the `alloc` statement. But we rejected this approach: in practice, clients need concrete control over headers and descriptors, because they are often used for additional purposes besides garbage collection. For example, our prototype Haskell compiler (Section 6) uses an additional descriptor field to encode the type of closures; similarly, object-oriented languages often use the header to point to a class descriptor record or vtable. Therefore, although GCminor requires that the record layout can always be determined from the record header, it is flexible about exactly how this connection is made. A record returned by `alloc` always contains a header, but the mutator is responsible for writing the header contents explicitly; if headers point to auxiliary static descriptors, the mutator must provide explicit global data definitions for those descriptors.

In general, therefore, the client must specify its desired layout description scheme to our system. Abstractly, the necessary information consists of two functions:

size : memory → value → nat
ptrP : memory → value → nat → bool

where *size* M h gives the total length (in words) of the record with header h in memory M, and *ptrP* M h n returns *true* if and only if field n of the record (numbered from 0) with header h in memory M contains a heap pointer. Concretely, the system requires

i. Cminor code macros `size` and `ptrP` that implement *size* M and *ptrP* M within the collector code.

ii. A Coq logical predicate
layout_desc : memory → value → nat → (nat → bool) → Prop

[2] Our proofs do not currently cover floats, because the Compcert v1.4 memory model does not permit us to write a collector that manipulates ints and floats uniformly as raw bytes.

Source code (Haskell notation, but strict):

```
f xs a = case xs of
           [] -> []
           x:zs -> (x+a):(f zs a)
```

GCminor code:

```
functions f = fun (xs,a) {
   stack 0; vars x,y,ys; roots xs,y,ys;
   if xs = null {
     return null;
   } else {
     ys = call(f,[load(int32,xs+4),a]);
     x = load(int32,xs);
     y = alloc(2);
     rstore(int32,y,-4,cons_header);
     rstore(int32,y,0,x+a);
     rstore(int32,y,4,ys);
     return y;
   }
}; ...
vars cons_header = {int32 2, int32 1}; ...
```

Figure 5. Possible GCminor code for a simple function over lists.

where *layout_desc* $M\ h\ s\ p$ holds exactly when s = *size* $M\ h$ and p = *ptrP* $M\ h$. The formal semantics of GCminor is parameterized by this predicate.

iii. A Coq lemma showing that `size` and `ptrP` are consistent with *layout_desc*, and that they are invariant under changes to the heap; the correctness proof for the collector is parameterized over this lemma.

Standard scheme. To let clients use our system without doing additional proofs, we provide a standard instantiation of these components for a particular, simple layout descriptor scheme. In this scheme, field order in records is constrained so that all *Atomic* values come before any *Ptr* values. (This ordering is convenient for describing closure records and simple class-based objects without inheritance.) Record layouts can thus be described by a simple two-element descriptor: the first element gives the total number of words in the record, and the second gives the number of words in the atomic prefix. We store these descriptors in global static memory; each record header is a pointer to such a descriptor. Figure 4 gives an example. The corresponding Cminor GC macro implementations are

```
size h := load(int32,h)
ptrP h n := load(int32,h+4) <= n
```

and the logical predicate is defined by

$$\frac{\textit{Mem.load int32 M b o = Some (Vint s)} \quad \textit{Mem.load int32 M b (o+4) = Some (Vint a)}}{\textit{layout\_desc M (Vptr b o) s } (\lambda n.a \leq n)}$$

Example Figure 5 shows an example of GCminor code, such as might be produced for a simple recursive function over integer lists. We assume that lists are represented by two-word cons cells in the usual way, with the empty list represented by the `null` pointer. The global `cons_header` gives the layout of a cons cell using the standard layout descriptor scheme.

Formal semantics. GCminor's small-step transition rules have the form

$$G \vdash_{GCM} (F, st, E, k, \sigma, M, A) \xrightarrow{t} (F', st', E', k', \sigma', M', A')$$

$$\frac{\begin{array}{c}\vdash M_1 : A \\ \textit{enough\_mem } R\ M_1\ s \\ R = \textit{env\_root\_values } F.\textit{roots } E \cup \textit{cont\_root\_values } k \\ (M_2, b) = \textit{Mem.alloc } M_1\ (-4)\ (4 \cdot s) \\ E' = \textit{env\_clear\_non\_roots } A\ F.\textit{roots } E \\ k' = \textit{cont\_clear\_non\_roots } A\ k \\ E'' = E'\{x \mapsto \textit{Vptr b 0}\} \\ A' = A \cup \{b\}\end{array}}{G \vdash_{GCM} (F, x \texttt{ = alloc(}s\texttt{)}, E, k, \sigma, M_1, A) \xrightarrow{\epsilon} (F, \texttt{skip}, E'', k', \sigma, M_2, A')}$$

Figure 6. GCminor `alloc` semantics transition rule

where most of the components are the same as in the rules for Cminor (see Section 2). In addition to global data and stack frame blocks, the memory M now contains a block for each heap record allocated so far. The set of these blocks is recorded in the new state component A. Continuations k now record the root sets as well as the environments of suspended function activations.

Representing each heap record by an entire fresh memory block is essential to abstraction. It makes it impossible to forge a pointer to a record, and prevents order comparisons (e.g., $\leq$) between pointers into different records. The former ensures that records unreachable from roots are truly inaccessible, while the latter allows GCminor to hide any movement of records that occurs in the implementation. Neither would be possible in a conventional flat memory model.

As noted above, it is the mutator's responsibility to initialize the header and fields of allocated records consistently, so that every field designated in the header as a *Ptr* contains a valid pointer into the set of heap records A (or is `null`). This well-formedness property is captured by the predicate *heap_record_ok*, defined as:

$$\frac{\begin{array}{c}\textit{Mem.load int32 } M\ b\ (-4) = \textit{Some } h \\ \textit{layout\_desc } M\ h\ s\ p \\ \textit{is\_atomic } A\ h \\ \textit{Mem.high\_bound } M\ b = 4 \cdot s \\ \textit{fields\_ok } A\ M\ b\ s\ p\end{array}}{\textit{heap\_record\_ok } A\ M\ b}$$

Here the *is_atomic* clause asserts that the header h is not itself a heap pointer, and *fields_ok* asserts that each field in record b has the correct GC type according to the pointer map p.

The mutator has some flexibility in initializing records, but it must ensure that every record is well-formed at any potential collection point, i.e., at any execution of an `alloc` (recall that we do not support incremental or concurrent collection). This is reflected in the semantics rule for `alloc`, shown in Figure 6; for simplicity, we give a version that allocates just one record at a time. This rule relies on a number of auxiliary predicates, which we describe in the remainder of this section. We write $\vdash M : A$ as an abbreviation for $(\forall a \in A, \textit{heap\_record\_ok } A\ M\ a)$, i.e., the entire heap A is well-formed in M. Note that a well-formed heap is necessarily *closed*: i.e., each *Ptr* field in each heap record points to some other heap record.

The *enough_mem* clause asserts that in the current program state there is still enough space to add a record of size s to the heap. The definition of this predicate depends on the style of memory manager being used. For a compacting collector and bump-pointer allocator, we can use the following definition, where the maximum allowed heap size is a symbolic parameter of the overall semantics.

$$\frac{\vdash M : A' \quad \forall b, (\mathit{Vptr}\ b\ 0) \in R \Rightarrow b \in A' \quad \mathit{sizeof}_M(A') + s \leq \mathit{maximum\_heap\_size}}{\mathit{enough\_mem}\ R\ M\ s}$$

This predicate doesn't actually compute the live heap; instead, it just asserts that there exists *some* well-formed, and hence closed, heap A' that contains all the root values and is small enough to permit the desired allocation. In fact, A' will be a subset of the A in the `alloc` rule, but we don't need to use this fact explicitly. The root value set R is calculated as the union of the root values in the current environment (*env_root_values*) and for any environments stored within the current continuation (*cont_root_values*).

If *enough_mem* does not hold, the `alloc` rule is not enabled, and the program gets stuck. (An alternative approach would be to add a companion rule stating that when there is insufficient memory, the `alloc` statement issues a runtime error message and enters an infinite loop representing a fatal exception. Unfortunately, it would be difficult to preserve these semantics through the remainder of the compilation pipeline, because subsequent compiler transformations can actually decrease the size of the live heap, so some allocations that fail at the GCminor level would succeed in the generated code!)

Mem.alloc allocates a fresh block in Compcert's underlying memory model, giving it appropriate lower and upper offset bounds (in bytes, and allowing for the header). A pointer to the first data word of the resulting block is assigned to x.

As noted above, GCminor hides any heap and environment changes to reachable pointers caused by a relocating collector. However, in the formal semantics, collection *does* have an observable effect on any *non-root* variables containing heap pointers: it clears them (i.e., sets them to the value *Vundef*). To keep the GCminor semantics deterministic, pointers are cleared at *each* `alloc` operation, both in the current environment (*env_clear_non_roots*) and in any environments stored within the current continuation (*cont_clear_non_roots*).

Well-behaved GCminor programs that specify their roots correctly will never observe pointer clearing, but ill-behaved programs that attempt to dereference a pointer fetched from a non-root variable will get stuck. For example, in the code of Figure 5, if we had omitted the declaration of `xs` as a root, the value of `xs` would have been cleared by an `alloc` within the recursive call, and the subsequent `load` into `x` would have gotten stuck. Since our semantics preservation proof for the GCminor-to-Cminor translation only needs to hold for non-stuck programs, it can ignore programs that mis-specify roots, which is essential to making the proof work. Of course, GCminor's actual implementation doesn't clear the non-accessible roots; this would be pointless, since correct programs wouldn't be able to tell the difference anyway.

4. GCminor Implementation

GCminor is implemented by translation to Cminor. This translation involves two key steps:

i. GCminor `alloc` statements are translated into calls to a fixed library function, written in Cminor, that performs the allocation after garbage collecting if necessary. (This call could be inlined for efficiency.)

ii. Code is inserted around each function call (including allocation calls) to save and restore live roots into in-memory stack frames. If the collector is invoked, it examines this data structure to find roots.

The remainder of the GCminor language is essentially identical to Cminor, so its translation is trivial.

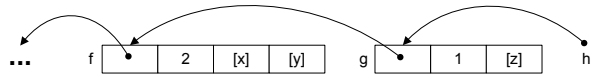


Figure 7. Stack layout example. We suppose that `f`, with two root variables `x` and `y`, calls `g`, with one root variable `z`, and that `g` in turn calls `h`.

Allocation. Translating `alloc` statements to calls is straightforward. Although the translation does not commit to a specific collection method, it does assume a bump-pointer allocator (such as in a Cheney collector). With this kind of allocator, it is more efficient for the mutator to make a single large allocation request than a series of small ones, so the translation of a multi-record `alloc` sums the requested sizes, requests a single record, and then updates the target variables with appropriate offsets into the resulting record.

Roots. The translation of root declarations is more complex. The fundamental difficulty is that the collector must be able to find—and, for a moving collector, also update—all roots for *all* functions suspended on the current call stack. But Cminor provides no direct access to local variables in suspended activations (unsurprisingly, since C doesn't require such a feature).

Our solution is to use a "shadow stack" [9, 15] in which live root values are stored in memory across calls. Specifically, the translation generates code to dump the local live roots to the stack before each call and restore them after each return. Roots are stored in a record in the function's explicit stack frame. Each root record is linked to that of its caller, so that the entire chain of root records can be traversed given a pointer to the most recent record, which is passed as an extra argument to every `call` (but not to `extcalls`). Figure 7 shows an example of the stack layout at the Cminor level.

As an important optimization, we store only *live* roots in root records. GCminor constrains root variables to contain valid roots at all times, so it would be safe to record *all* roots, but this could prevent some garbage from being collected. We also have some minor optimizations for cases when no local variables need to be stored on the stack.

LowGCminor. To subdivide the implementation and proof effort, we introduce LowGCminor, a further intermediate language between GCminor and Cminor. LowGCminor is syntactically identical to GCminor, except that there is no per-function list of roots; instead, each `alloc` statement and non-tail call to an internal function has an additional component listing the set of live roots at this particular site. This set can be thought of as an abstract form of the GC root tables used by many collectors. As in GCminor's semantics, any heap pointers omitted from the root set are (conceptually) cleared by a collection.

Cheney collector. Our actual GCminor implementation currently uses a simple Cheney-style copying collector.[3] The collector uses two large fixed semi-spaces declared in Cminor's global data region. The allocation pointer and the limits of the current and reserve spaces are also held in globals.

Other collector architectures. Changing to a different collector would have only modest impact on the structure of the GCminor-to-Cminor translation. A collector (e.g., mark-sweep) that doesn't move records still needs to *read* roots from all suspended functions, but doesn't need to *change* them. For such a collector, it would be unnecessary to restore values from the root records after returning from calls. For an allocator that uses free lists rather than pointer

[3] We thank Xavier Leroy for providing collector code on which ours is closely based.

bumping, we would want to generate separate allocation requests for each record in a multi-allocation. For a generational collector, the translation would need to generate suitable write barrier code at `rstore` statements.

5. Semantics preservation

Our overall semantics preservation proof for the GCminor-to-Cminor translation is conducted in the same style as the existing Compcert proofs. It assumes correctness of the allocation function with respect to a low-level specification. The proof has two parts, bridged by LowGCminor. We describe each part in turn, and then discuss the allocator specification and how it can be realized.

LowGCminor. The essence of the translation from GCminor to LowGCminor is an analysis that computes the liveness of root variables at each point where a root record must be constructed. The proof that the translation preserves semantics must show that no live pointers are omitted, i.e., that the liveness analysis is correct.[4] The simulation relation for the proof is very simple: GCminor state S_1 and LowGCminor state S_2 are related only if they are identical except for their local environment components E_1 and E_2, which need only agree on the live variables L computed for that state:

$$E_1 \sim_L E_2 ::= \forall x \in L.\, \forall v.\, E_1(x) = v \rightarrow E_2(x) = v$$

For technical reasons, LowGCminor requires that values written to root records be valid pointers. The proof of this invariant follows easily from GCminor's invariants and from the insertion of explicit `null` initializers for variables that are used before they are assigned. The simulation proof is particularly simple because each LowGCminor program contains exactly the same statements as its GCminor original, except for the added initialization code.

Cminor. The semantics preservation proof for LowGCminor to Cminor is much harder. First, the structure of source and target code differs substantially at allocations and call sites, where the Cminor code introduces new statements to invoke the allocation library function and to copy root variables to and from the shadow stack explicitly. The invariant relation between LowGCminor and Cminor states must account for the addition of explicit root records in stack frame memory at the Cminor level, and extra parameters and local variables in the environment. Furthermore, because we reason using a precise model of machine arithmetic, we have the burden of proving that none of the generated code (such as the save and restore code) causes arithmetic overflow.

Second, and more fundamentally, there is a major change in the representation of the heap. In (Low)GCminor, each allocated heap record is represented by a pointer to an independent memory block, and these records appear never to move. In Cminor, the actual implementations of the allocator and collector are exposed. For example, if a Cheney copying collector is used, records are represented by pointers into the middle of a single large block that holds the entire current semi-space, and they can move at any collection.

The invariant relation between (Low)GCminor memory and Cminor memory is thus a dynamic isomorphism over live blocks, described by a memory embedding ϕ that maps GCminor addresses to Cminor addresses. Initially, ϕ is empty. It is extended every time an object is allocated: at the GCminor level the object is stored in a fresh block, while at the Cminor level it is stored at a free location in the heap block. When a collection occurs, the old embedding can be composed with a partial map describing the movement (and possible freeing) of each object to create a new embedding.

[4] Our current proof makes this easy by omitting support for `loop`s, which our example front end doesn't need.

$$\frac{\begin{array}{c}(R, M_1, A) \sim_\phi (P, M_1') \\ \vdash M_1 : A \\ \mathit{enough\_memory}\ R\ M_1\ s \\ \forall v, v \in R \Rightarrow \mathit{is\_pointer}\ v\end{array}}{\begin{array}{c}\exists M_2', b', \mathit{ofs}', \phi'. \\ \left(\begin{array}{c} G \vdash_{\mathtt{CM}} (F, x\ \mathtt{=\ call(}\mathit{alloc}\mathtt{,[}P\mathtt{,}s\mathtt{])}, E, k', \sigma, M_1') \xrightarrow{\epsilon} \\ (F, \mathtt{skip}, E\{x \mapsto \mathit{Vptr}\ b'\ \mathit{ofs}'\}, k', \sigma, M_2') \end{array}\right) \\ \wedge\ (R, M_1, A) \sim_{\phi'} (P, M_2') \\ \wedge\ \mathit{free\_block}\ \phi'\ M_1\ M_2'\ b'\ \mathit{ofs}'\ s \\ \wedge\ \mathit{stack\_preserve\_mem}\ k'\ m_1'\ m_2'\ P \\ \wedge\ \mathit{nobj\_preserve}\ \phi\ \phi'\end{array}}$$

Figure 8. Cminor-level allocator specification, with roots in the shadow stack P.

To manage this part of the proof, we factor out the behavior of the allocation function into an abstract specification, which is then refined several times until we reach a low-level description specialized to a Cheney collector. We have proved correctness of the translation relative to each refinement level of this specification.

Allocator specification. The allocator specification at each refinement level describes the behavior of the Cminor-level *alloc* function call corresponding to the GCminor allocation in Figure 6. The *alloc* function may choose to perform a GC; the assumptions of the specification ensure that the GC will not crash when run, while the conclusion describes the state after the allocation is successful.

At the highest level, the *alloc* specification describes the effect of collection on local variables in the current environment and continuation. For brevity, we avoid describing this specification level and instead concentrate on the next refinement level, where roots are assumed to be stored in the shadow stack. This specification, given in Figure 8, is the most important part of the interface between GCminor and the GC. This style of specification is abstract enough to describe a range of collectors, including those that move or coalesce heap records, and has been used successfully in prior work [14, 22, 24] to verify copying, mark-sweep and incremental copying collectors.

The core of the specification is the simulation relation $\sim_\phi$. The initial GCminor state (represented by the root values R, GCminor memory M_1 and set of objects A) must be related to the initial Cminor state (the shadow stack, viewed as a list of root frame addresses P, and the Cminor memory M_1') via the embedding ϕ, written as $(R, M_1, A) \sim_\phi (P, M_1')$. This relation states that the the root values in R are represented in a linked list of arrays with nodes given by P, which ensures the correctness of root restoration. It also requires that M_1 is embedded in M_1' via ϕ, without overlapping in memory. The precise definition of the memory embedding depends on the collector being used, and will include the private data needed by the GC. The other preconditions come directly from the GCminor semantics (the heap must be well-formed and there must be enough free memory) or from LowGCminor (all roots must be valid).

The first part of the postcondition asserts that the call to the function *alloc* will succeed and return to the state including memory M_2'. *alloc* is called with two arguments, a pointer to the linked list of saved roots (the first element of P) and the number of words to be allocated, s. The allocation function will return a pointer $\mathit{Vptr}\ b'\ \mathit{ofs}'$ to a fresh record $4s$ bytes long, by setting the local variable x to the start of the record.

After the collection, the embedding ϕ has changed to ϕ'. However, the same fragment of GCminor state (R, M_1 and A) is represented in the new Cminor memory M_2'. From the client's perspective, this means that the roots and memory have not changed.

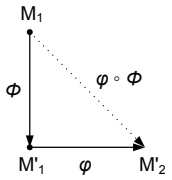
From the collector's perspective, the use of a new embedding allows records to be moved at the Cminor level.

The *free_block* predicate states that there is enough unallocated space at address *Vptr* b' *ofs*$'$ to hold s words of memory. The definition of this predicate depends on the collector being used. Taken together with the $\sim_{\phi'}$ injection, *free_block* ϕ' M_1 M_2' b' *ofs*$'$ s ensures that when we extend M_1 with a fresh object block b to produce a new memory M_2 at GCminor level (see Figure 6), we will be able to extend ϕ' to a fresh embedding of M_2 into M_2' that maps b to the address *Vptr* b' *ofs*$'$.

Finally, the specification ensures that the GC does not damage other parts of memory. The predicate *stack_preserve_mem* states that the length and content (aside from the saved roots) of Cminor stack frames must be unchanged from M_1' to M_2'. This ensures that the collector does not change the portion of the stack frame visible at the GCminor level. The predicate *nobj_preserve* states that the GC does not move any non-records (i.e., stack frames and global memory).

Cheney collector In this section we describe the further refinement of the *alloc* specification for a typical Cheney copying collector, by defining ϕ and *free_block*. A Cheney collector stores all objects in a single semispace block *objb*. A free pointer *free* points to the next unused location in *objb* and the limit pointer *limit* points to the end of *objb*. The injection ϕ maps GCminor blocks that contain objects to offsets within *objb*, and other GCminor blocks directly to Cminor blocks (disjoint from *objb*). For this injection to be well-formed, no two pieces of GCminor memory can be mapped onto a single piece of Cminor memory. For *free_block*, a Cheney collector requires that *free* $\leq$ *Vptr* b' *(ofs*$' - 4$*)* and that *Vptr* b' (*ofs*$' + 4s) \leq$ *limit*.

Cheney collection causes reachable objects to be copied to a new semispace block. Generating the new injection ϕ' after a collection is the key part of the preservation proof. At the concrete level, the movement of objects by the collector can be given by an isomorphism φ from the initial location of a reachable Cminor object to the final location of that object. The mapping ϕ' from GCminor objects in M_1 to the new Cminor objects in M_2' can then be defined as the composition of the old mapping ϕ with φ, as shown by the following diagram:



In other words, if b is a reachable GCminor object, then $\phi'(b) = \varphi(\phi(b))$. $\phi(b)$ produces the initial location of the concrete representation of b, and φ produces the final concrete location of that object. If b is unreachable (and thus was not copied by the collector), then $\phi'(b)$ is undefined.

Low-level collector proof. We have partially verified the safety and completeness of our Cheney copying collector (Section 4) written in Cminor using separation logic tactics [21], but have not yet formally connected this separation-logic specification to the one given in Fig. 8. In order to connect this proof to the rest of our system, we must show that the concrete specification of the collector matches the abstract specification given here, verify termination of the collector, and formally relate the separation-logic proof system to the style of specification shown here (where separation facts must be made explicit).

6. Case Study: Haskell to Dminor to GCminor

To assess the utility of GCminor as a compilation target for an existing programming language, we have built a prototype compiler for a subset of Haskell [26]. Our compiler supports most of the features of Haskell 98, except for floating point, file I/O, arrays, and `seq`.

Compiling Haskell to a low-level, call-by-value language such as GCminor involves a number of major program transformations, including an implementation of lazy evaluation using `force` and `delay` constructs [1, p. 261], conversion of higher-order functions and delayed thunks to first-order functions and closure records [4], and conversion from an expression-based, purely functional form to a statement-based, imperative one. Although a number of compilers have compiled Haskell by transformation to a strict language with explicitly lazy constructs [5, 6, 11, 12], currently GHC instead relies on specialized runtime system support for laziness [27]. Demonstrating that our minimalist runtime system is adequate to run Haskell programs is an important step towards increasing the overall assurance of Haskell-based applications.

The starting point of our compilation pipeline is External Core [35], a text-based representation of code in GHC's intermediate language Core [28]. Core is based on System F_C, which is an implicitly lazy language that extends System F with algebraic data types, a `let` construct, and type equality coercions [30]. Our compiler takes as input a Core program that has already been heavily optimized by GHC's front end. It then passes the program through the transformations described above, each of which produces a program in a different call-by-value intermediate language. These transformations expose further opportunities for standard functional language optimizations such as uncurrying, let-floating, identifying functions that do not require closures, and inlining, as well as removal of redundant `force` and `delay` operations. Finally, the pipeline produces a GCminor program.

Memory safety. In addition to demonstrating that GCminor is a reasonable compilation target for a sophisticated high-level source language, this prototype illustrates how our framework can provide useful assurance guarantees short of full semantics preservation. Building a fully semantics-preserving compiler accepting Haskell source would be a daunting task (even if there were a generally-accepted formal semantics for Haskell), especially since we would need to prove the correctness of the optimizations that GHC applies to Core. Instead, we lay the groundwork for an assurance argument based on a *combination* of semantics preservation proofs and weaker, but much easier, type soundness proofs. Specifically, we select one of our intermediate languages, called Dminor, to serve as the boundary between the two kinds of proofs.

Dminor has a type system, which is designed to be *sound*: well-typed programs don't get stuck.[5] Semantics preservation for the remainder of the pipeline guarantees that a non-stuck Dminor program yields a non-stuck assembly language program. Finally, Compcert's definition of assembly language semantics (Section 2) implies that a non-stuck program is memory-safe, in the sense that it cannot forge pointers or dereference memory outside of properly allocated stack frames or global memory regions. Combining these properties yields a memory safety guarantee for the entire back end. By typechecking the Dminor code generated by our front end, we obtain a memory-safe Haskell compiler with "fail-stop" behavior: any program that compiles successfully will be memory-safe. In fact, we have also implemented type-checkers for Core and our other intermediate languages, which we use to check type-correctness at each compilation stage; this technique is an excellent

[5] Strictly speaking, even a well-typed program might get stuck unless the Dminor analogue of *enough_mem* holds at every allocation point.

way to find compiler bugs. Of course, if the front end is bug-free, it should always generate well-typed Dminor code. We lack machine-checked soundness proofs for the type systems of our languages, but we believe that doing these proofs would be straightforward. We also expect that the various transformations and optimizations performed along the pipeline all preserve types, although for the most part we do not have formal proofs. We are confident that these type preservation results could, with sufficient effort, also be proved within Coq, obviating the need for compile-time typechecking of the intermediate forms.

Minimalist type system. Unfortunately, while proofs about types are typically much easier than proofs of semantics preservation, they may still be quite hard if the type systems involved are complex. Standard type systems that can typecheck closure-converted code and record initialization require features such as existential types and initialization types [23]. Explicit `force`s require strong updates, which would add yet more complexity to our type system. Moreover, in our compiler, these features would need to be *added* to the already very complex System F_C. We therefore adopt a different approach, and give the intermediate languages in our pipeline extremely rudimentary type systems that serve only to distinguish which local variables hold heap pointers and to describe the pointer layout corresponding to each heap record tag. These type systems are about as minimalist as they can be while still supporting static identification of roots at the GCminor level.

This approach simplifies both engineering of type checkers and proofs involving types. However, our type systems are so weak that they can neither distinguish function closures from ordinary data records nor track which algebraic data type a data record belongs to. If our front end had bugs, it could produce Dminor programs that tried to apply data records as functions or inspect function closures with `case` expressions; yet, these programs would still be well-typed. Thus, in order to prove the type systems sound, we must add additional runtime checks at closure applications and some record accesses, so that these nonsensical programs yield a checked runtime error. Checks never fail in code generated by a bug-free compiler, but they may increase execution time even so. We think that this idea of trading off verification complexity against runtime performance has some merit, but we stress that the use of dynamic checks is quite independent of the remainder of our framework.

Dminor syntax and semantics. Dminor (Figure 9) is a low-level, first-order, strict, expression-based, pure functional language with a very simple type system. Its design is a compromise between ease of typability and simplicity of translation. Including closure application and thunk forcing as primitive operations in the language facilitates typability, while using almost the same set of underlying pure operations as GCminor simplifies translation.

The language divides expressions into two categories: pure and monadic [13]. Pure expressions correspond directly to GCminor expressions, with the addition of a separate `rload` operator to read from heap objects (which avoids the need to type address arithmetic) and the omission of $\texttt{stackaddr}_i$ (as there are no explicit stack frames). Monadic expressions may have effects. They can appear only in tail position or in a `let`; all other subexpressions must be pure. This syntactic structure sequences effectful operations explicitly and simplifies subsequent translation to GCminor.

Dminor supports just three type constructors: unboxed integers (`Int`), pointers to heap records with unknown tag ($\Box$, pronounced "box"), and pointers to heap records of known tag, written $\langle c \rangle$ where c is a constructor tag (explained below). Any value of type $\langle c \rangle$ can be statically coerced to type $\Box$. In code translated from Core, most variables have type $\Box$; only variables explicitly declared as unboxed integers (type `Int#` in GHC) or pointers to static global memory such as string literals (type `Addr#` or `Ptr` in GHC) are given type `Int`. Fortunately, GHC does not permit polymorphism over unboxed values, so the boxity of a value is always apparent. We introduce record types while doing closure conversion and through static analysis to make the types of certain $\Box$-typed values more precise (eliminating the need for some `case` expressions). All bindings are statically typed, making it possible to compute the type of any expression without an environment. No identifier can be bound twice in the same function; this simplifies formalization of the semantics and eases translation to GCminor, which lacks nested scopes.

$$
\begin{array}{rcll}
t & ::= & \texttt{Int} & \text{integer} \\
 & | & \Box & \text{pointer to record} \\
 & | & \langle c \rangle & \text{pointer to record with tag } c \\
\\
e & ::= & \ldots & \text{(as in GCminor)} \\
 & | & \texttt{rload}(ch, e_{record}, e_{offset}) & \text{load from record field} \\
 & | & \sout{\texttt{stackaddr}_i} & \sout{\text{address of stack frame entry}} \\
\\
m & ::= & e & \text{pure expression} \\
 & | & \texttt{app}(e, \vec{t} \to t, \vec{e}) & \text{closure application} \\
 & | & \texttt{call}(f, \vec{e}) & \text{internal function call} \\
 & | & \texttt{let } [id :: t \texttt{ =] extcall}(f, \vec{e}) \texttt{ in } m & \text{external call} \\
 & | & \texttt{let } id :: t \texttt{ = force } e \texttt{ in } m & \text{thunk evaluation} \\
 & | & \texttt{letrec } \overrightarrow{id = c(\vec{e})} \texttt{ in } m & \text{record binding} \\
 & | & \texttt{case } e \texttt{ of } \overrightarrow{id :: \langle c \rangle : m} & \text{case analysis} \\
 & | & \texttt{if } e \texttt{ then } m \texttt{ else } m & \text{conditional} \\
 & | & \texttt{let } id :: t \texttt{ = } m \texttt{ in } m & \text{monadic binding} \\
\\
f & ::= & \texttt{fun}(\overrightarrow{id :: t}) \ \{ \ \texttt{vars } \overrightarrow{id :: t}; \ m \ \} & \\
\\
p & ::= & \texttt{functions } \overrightarrow{id = f}; & \\
 & & \texttt{tags } \overrightarrow{c \mapsto \vec{t}}; & \\
 & & \texttt{closuresigs } \overrightarrow{c \mapsto (\vec{t} \to t)}; & \\
 & & \texttt{vars } \overrightarrow{id = initializer}; & \\
 & & \texttt{main } id &
\end{array}
$$

Figure 9. Syntax of Dminor types (t), monadic expressions (m), functions (f), and programs (p). Atomic expressions (e) are the same as for GCminor, except as noted. Record constructor tags (c) are described in the text.

The semantics of most Dminor expressions are standard or similar to their GCminor equivalents; we describe those that are not. All memory is allocated by `letrec` expressions. Evaluating

$$
\begin{array}{ll}
\texttt{letrec} & x_1 \texttt{ = } c_1(e_{11}, \ldots, e_{1p_1}) \\
 & \ldots \\
 & x_n \texttt{ = } c_n(e_{n1}, \ldots, e_{np_n}) \\
\texttt{in } m &
\end{array}
$$

simultaneously allocates n records such that record j has tag c_j and fields given by the values of pure expressions $e_{j1}, \ldots, e_{jp_j}$. The bindings are recursive in the sense that the x_i (but *not* loads from them) may be mentioned in the e_{jk}.

Closures and thunks (which are simply closures taking zero arguments) are constructed just like ordinary data records; the only difference is that the signature $\vec{t} \to t$ of the closed-over function is included in the `closuresigs` list. The semantics of `force` reflect Haskell's call-by-need semantics: forcing a thunk means evaluating it, then overwriting the pointer to the thunk with a pointer to the result. The latter step means changing a record's tag, an operation which would have complicated Dminor's type system if expressed explicitly in the language.

Every record, whether it is a closure record or a data record, has two header fields (meaning that the size of every record incorporates two extra words in addition to the sizes of its fields). One

word—the record tag—is, abstractly, an index into the list `tags` of record layouts. The tag plays two roles for any given record: first, it points to layout information that the collector uses, as described in Section 3; second, if the record is a data record, then a *case* expression that deconstructs it will check its tag, and if the record is a closure record, then an *app* expression that applies it will check its tag to obtain the closure's type signature. The second word contains the code pointer for closure records and goes unused for data records (to obtain uniformity, we pay an extra word per data record). The translation from Haskell to Dminor assigns a unique tag to each declared algebraic data type constructor and assigns a unique encoding to each possible closure type signature.

A `case` expression dispatches on the record's tag value. The type of a case discriminant x is normally $\Box$. Within an arm of the form $x' :: \langle c \rangle : m$, identifier x' is bound to the value of x and given the refined type $\langle c \rangle$; this allows fields of x to be accessed within the arm by `rload(`*ch*`,`x'`,`*offset*`)`, which is well-typed if *offset* and *ch* are valid for the record layout corresponding to c. Dminor's type system is not powerful enough to check that a `case` expression is exhaustive; a *case* expression in which no listed constructor matches the discriminant denotes a runtime error. As a result, the compiled code may test more alternatives than would be required if `case`s were known to be exhaustive.

In the expression `app(`*op*, $\vec{t} \rightarrow t$, *args*`)`, the operator *op* should evaluate to a pointer to a closure record, whose first field is a (top-level) function f, which is invoked with the values of arguments *args* and (implicitly) the closure record itself. The second field $\vec{t} \rightarrow t$ is the expected type signature of the function being applied. At runtime, the static signature is checked against the signature associated with the operator closure record's tag; if the record has an unexpected signature, or isn't a closure record at all, a runtime error is raised. We chose to keep closure application as a primitive operation in Dminor in order to make the language typable without introducing existential types or more runtime checks. Dminor also includes `call`s to known (top-level) functions; these do not require a runtime type check. By default, the translation from Haskell must compile function applications as closure applications, but static analysis can transform some of these applications to known-function calls.

Translation to GCminor. The translation of Dminor to GCminor is largely straightforward. Expressions must be converted to statements; the most complicated translation is from `case` expressions to nested `switch`, `block`, and `exit` statements, but this is similar to existing Compcert code for compiling C switch statements, so we omit further details here. Each `letrec` expression is translated to an `alloc` statement followed by a series of `rstore` statements to initialize the headers and fields. The scopes of locally-bound identifiers are widened to the entire function; this is safe because no identifier is bound twice in a Dminor function. Identifiers of type $\Box$ or $\langle c \rangle$ are declared as roots of the GCminor function.

Tags and closure signatures are made concrete as follows. Each tag *c* is converted to an offset *o* into a static global descriptor array.[6] We write *dbase* for the base of this array. At runtime, a record header contains a direct pointer to the descriptor (that is, *dbase*$+o$). The descriptor has three words: number of fields, number of atomic fields, and encoded closure signature type s (explained below). This format is compatible with the standard descriptor scheme described in Section 3. (The translation will reorder fields as necessary to keep atomic fields first.)

[6] For historical reasons, our current system actually uses a version of Dminor in which the concrete descriptor array is already present; the Dminor type checker is responsible for confirming that the abstract and concrete representations of tag and closure signature information agree.

Program	GT	GM	CT/GT	CM/GM
circsim	1.81	672	3.94	4.98
clausify	1.04	228	1.56	2.11
cryptarithm1	2.04	1029	1.46	1.68
cse	4.26	<1	5.42	1.71
gcd	11.89	5209	1.91	2.33
hartel_comp_lab_zift	1.09	405	1.98	1.97
hartel_ida	1.19	364	1.62	2.06
hartel_sched	3.31	1057	1.33	1.91
hartel_transform	1.87	738	1.78	1.86
hartel_typecheck	1.48	330	1.54	2.21
knights	4.62	170	0.88	4.59
lambda	3.16	634	1.82	2.10
last-piece	1.92	411	2.36	4.95
lcss	1.59	625	3.64	1.97
multiplier	1.55	464	2.50	1.76
power	41.1	13388	1.65	2.11
primetest	233	74750	2.82	3.62
rewrite	1.58	268	2.23	4.86
Mean			2.04	2.49

Figure 10. Comparing time and space usage for GHC and Haskell Compcert. For each program, the "GT" column shows its runtime in seconds and the "GM" column shows its allocation (in megabytes) when compiled by GHC. The "CM/GM" column shows the ratio of memory allocated by each Compcert-compiled program, compared to the GHC baseline. The "CT/GT" column shows the same ratio, but for time instead of memory. "Mean" shows the geometric mean ratios over all 18 programs.

The GCminor code generated for a `case` statement dispatches on the record's tag value, which must be retrieved from the header by subtracting *dbase*; it is impractical to dispatch directly on the descriptor pointer itself, because absolute descriptor addresses are not known at compile time. We must implement the dispatch as a binary comparison tree.[7] At the GCminor level, the function signatures that appear in `app` expressions and in the global descriptor array are represented by distinct integer encodings s, which can be cheaply compared for equality at runtime. These integers can be easily assigned by a global traversal of the program during translation.

We have a machine-checked proof of semantics preservation for most of the Dminor-to-GCminor translation, excluding only `case`, `app`, and `force` expressions, which do not interact with allocation in interesting ways. As usual, the key to a semantics preservation proof for the translation is the simulation between Dminor and GCminor states. Although the relationship between Dminor and GCminor continuations is complicated, the heap memory components are identical. Thus, the crucial proof obligation induced on clients by the GCminor semantics, namely that $\vdash M : A$ at each allocation point, can be proved as an invariant of Dminor in isolation. In fact, we prove that the heap is well-formed after *every* possible Dminor evaluation step.

Practical experience. To assess the practicality of our Haskell Compcert pipeline, we ran it on a number of benchmarks from the "spectral" section of the Haskell `nofib` benchmark suite. The "spectral" benchmarks are "small, key pieces from real programs" [25]. In order to compile real Haskell code, we had to make some changes to GHC's standard libraries to circumvent code that is based on primitive operations we have not implemented. In par-

[7] Jump tables, which require choosing the tags for each algebraic type from a dense domain of small integers, may be a more efficient implementation. But our choice of globally unique record tags forbids us from using them, and our version of Compcert does not support them in any case.

ticular, we reimplemented I/O functions as foreign calls to functions implemented in our simple RTS. In addition, we recompiled GHC with a native Haskell version of the multi-precision integer library [32] (substituting for GMP). We started with a baseline of GHC version 6.10.4.

Of the 60 spectral benchmarks, we chose 18 to present in this paper. We excluded benchmarks that ran for less than 1 second when compiled by Haskell Compcert, as well as benchmarks that used Haskell or GHC features we do not support. We used our patched version of GHC to generate Core programs as input to Haskell Compcert, as well as to generate baseline executables. We ran GHC with the `-O2` and `-fvia-C` flags, and ran all programs with a 128 MB heap. We did the measurements on an Apple Xserve G5 (2.3 GHz PowerPC dual core, 8 GB RAM, Mac OS X Server 10.4.11).

Figure 10 compares the performance of GHC-compiled and Compcert-compiled programs. (In both cases, time measurements included both mutator time and GC time.) On average, Compcert-compiled programs ran about twice as slowly as GHC-compiled programs, and also allocated 2.5 times as much memory. We can explain some of the memory allocation overhead by reference to our inefficient record layout (as described earlier in this section) and our strategy for compiling laziness. Also, our thunk representation introduces an extra three-word record for every thunk the program allocates. And in order to avoid implementing multiple return values in the back end, we compiled GHC's unboxed tuples by transforming them into boxed tuples. Given these pervasive sources of overhead, a mean factor of 2.5 increase in memory usage is not surprising.

Since allocation is expensive in our system (invoking the allocator requires function calls), a corresponding increase in execution time is not surprising either, but in fact time and memory overheads of individual benchmarks are often not well-correlated. Indeed, the sources of time overhead are still somewhat mysterious to us. One obvious possible source is the cost of maintaining the shadow stack of GC roots. To test this possibility, we selected a subset of benchmarks that do not require GC when run in a 1GB heap (the largest we can configure). Removing the shadow stack management code from these benchmarks improved their mean execution time by less than 3%, with little variance among the programs. Runtime type checks are another potential source of overhead. But compiling our benchmark set without runtime checks had almost no effect on mean execution time, although it did reduce the time of one benchmark (`power`) by 17%. Of course, the overhead of checks would increase if we eliminated other sources of overhead and thus reduced overall execution time. Another obvious point is that our garbage collector is slow and simplistic compared to GHC's highly tuned generational collector, but again, higher execution time overheads are not well-correlated with amount of GC performed. Investigating other possible sources of the performance gap remains as future work.

7. Related work

Dargaye [9, 10] extends the Compcert framework to compile miniML, a simple call-by-value functional language. The compiler uses a chain of new intermediate languages connected by semantics-preserving translations and ending with Cminor. Dargaye makes no attempt to present a general-purpose variant of Cminor for interfacing to a collector. However, the last of her new languages, Fminor, is quite similar to our Dminor (without support for laziness), though higher-level in some respects (e.g., `case` expressions bind constructor fields to identifiers) and lower-level in others (e.g., live roots are already explicitly identified, as in our LowGCminor). One significant simplification is that miniML contains no primitive types or operations; all values are boxed and all heap blocks have the same format (a single atomic constructor tag or closure function pointer, followed by value pointers) so there is no need for the front end to pass record layout information.

Dargaye's implementation of the memory management library is also quite similar to ours; we share similar collector code and shadow stack format, although she chooses to store roots in the shadow stack permanently rather than to store and reload them around calls. Like us, Dargaye axiomatizes the behavior of the allocation function. She explicitly defines reachability in the heap (made simpler because miniML heaps cannot have cycles of pointers), and specifies that collection should leave reachable memory locations unchanged. We use a similar but simpler specification at the GCminor level (*all* memory should remain unchanged), but at the Cminor level we refine it to a more precise specification that lets us describe the behavior of a moving collector.

McCreight et al. [20, 22] discuss the treatment of a garbage-collected heap as an abstract data type to hide the implementation details of a collector from the mutator, and verify in Coq that several collectors satisfy this interface. Hawblitzel and Petrank [14] apply this approach to realistic collectors for the Bartok C# compiler, using an automated theorem prover to verify the collectors. The root and record descriptor information needed by the collector-mutator interface is verified using a typed assembly language [7]. The final allocator interface of our work, given in Fig. 8, is also based on this approach. The main difference in our work is that it takes a *local* specification of parts of memory and wraps it up into a *global* specification in the form of a complete intermediate language, GCminor. This allows clients to reason about mutator programs at a single level of abstraction that hides the action of the collector.

Myreen [24] verifies a Cheney collector for a simple fixed record format, and uses a memory embedding to hide the movement of records from high-level code. However, the high-level state does not include any non-root components, so he does not have to deal with stale record pointers. Chlipala [8] carries out semantics-preserving compilation of the simply-typed lambda calculus to a low-level machine with garbage collection. He also uses an embedding from a high-level memory to a low-level memory, but does not hide the actions of the collector from the high level. Vanderwaart and Crary [37] use a type system to describe the interface to a precise collector. Their focus is on describing the layout of roots within the stack, and their work only supports reasoning about the type safety of the mutator code.

The C-- generic intermediate language [29] has a small runtime system with activation inspection primitives designed to support an arbitrary garbage collector provided by the compiler front end without the need for a shadow stack. It would be interesting to attempt a semantics preservation proof for a version of Cminor extended with primitives of this kind.

8. Conclusions and Future Work

We have described a general-purpose, machine-verified compilation pipeline for garbage-collected languages. A key feature of this system is the use of language abstraction to hide collection from the mutator. This is embodied in our language GCminor, which makes precise the often-subtle collector-mutator interface. Compiler writers can take advantage of our work simply by generating GCminor code—verifying that code to whatever level they desire—and then applying the existing Compcert back end to generate code for the PowerPC or ARM.

The work reported here represents a serious engineering effort stretching over several calendar years. The verified GCminor-to-Cminor compiler is about 13,000 lines of Coq code and proof scripts; the Core-to-GCminor front end is about 10,000 lines of

(heavily commented) Haskell and the Dminor-to-GCminor preservation proof is another 5500 lines of Coq scripts.

Our framework depends heavily on the existing Compcert system. Using Compcert has allowed us to build a working compiler quickly and has given us an excellent model for developing semantics preservation proofs for our extensions. We have remained fully backwards-compatible with Compcert, so that existing proofs are unaffected; indeed, we have changed only a very few existing Compcert files at all (in order to extend module signatures). This approach has caused a few problems: the Compcert memory model cannot express some useful GC techniques, and the lack of stack introspection requires using the awkward shadow stack technique. These limitations are not inherent in the Compcert framework, but removing them would be a significant task with possibly extensive ramifications for the existing proofs.

We have exercised our framework by building a compiler from GHC's Core intermediate language to GCminor. With simplicity of verification in mind, we designed intermediate languages for this compiler that use a novel combination of static and dynamic type checking. An alternative, which we plan to explore, would be to build a more conventional TAL-like type system [23] for GCminor, which would obviate the need for a Dminor-like higher-level typed language and corresponding semantics preservation proof.

Our performance measurements show that our Compcert-based compiler generates code that runs at about half the speed of GHC-generated code on average. We achieved this level of performance by combining an existing front end and back end, without extensive performance tuning. We conclude that it is possible to increase the assurance of high-level language compilers without seriously injuring performance.

Choosing a real language and compiler as a testbed has some obvious advantages, but also inevitably introduces a great deal of "accidental" complexity. We initially underestimated how hard it would be to understand the behavior of our back end on code generated by GHC's front end.

The verification story for our pipeline is almost complete. Our first priority for future work is to fill the remaining gap, which is between the allocator specification assumed by our GCminor-to-Cminor proof and the one we have proven for our prototype Cheney collector. We also plan to prove that the specification can be met by more realistic collectors, such as a generational copying collector.

Finally, our garbage collection framework is just one component of a larger effort to build a complete high-assurance runtime system suitable for supporting safety-critical and security-critical applications [33]. We hope to address other components, including concurrency and foreign function interfacing, in future work.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, first edition, 1985.

[2] ADT Coq. The Coq proof assistant. `http://coq.inria.fr`.

[3] A. W. Appel and S. Blazy. Separation logic for small-step Cminor. In *TPHOLs*, volume 4732 of *LNCS*, pp. 5–21. Springer, 2007.

[4] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *POPL*, pp. 293–302. ACM Press, 1989.

[5] A. Bloss, P. Hudak, and J. Young. Code optimizations for lazy evaluation. *Lisp and Symbolic Computation*, 1(2):147–164, 1988.

[6] U. Boquist and T. Johnsson. The GRIN project: A highly optimising back end for lazy functional languages. In *IFL '96*, volume 1268 of *LNCS*, pp. 58–84. Springer, 1996.

[7] J. Chen, C. Hawblitzel, F. Perry, M. Emmi, J. Condit, D. Coetzee, and P. Pratikaki. Type-preserving compilation for large-scale optimizing object-oriented compilers. In *PLDI*, pp. 183–192, 2008.

[8] A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *PLDI*, pp. 54–65. ACM, 2007.

[9] Z. Dargaye. MLCompCert Coq proofs. `http://gallium.inria.fr/~dargaye/mlcompcert.html`, 2009.

[10] Z. Dargaye. *Vérification formelle d'un compilateur pour langages fonctionnels*. PhD thesis, Université Paris 7 Denis Diderot, July 2009.

[11] A. Dijkstra, J. Fokker, and S. D. Swierstra. The architecture of the Utrecht Haskell Compiler. In *Haskell Symp.*, pp. 93–104. ACM, 2009.

[12] K.-F. Faxén. *Analysing, Transforming and Compiling Lazy Functional Programs*. PhD thesis, Royal Institute of Technology, June 1997.

[13] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*, pp. 237–247. ACM, 1993.

[14] C. Hawblitzel and E. Petrank. Automated verification of practical garbage collectors. In *POPL*, pp. 441–453. ACM, 2009.

[15] F. Henderson. Accurate garbage collection in an uncooperative environment. In *MSP/ISMM*, pp. 256–263, 2002.

[16] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pp. 42–54, 2006.

[17] X. Leroy. The Compcert verified compiler. `http://compcert.inria.fr/doc/index.html`, April 2009.

[18] X. Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, 2009.

[19] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *J. Autom. Reason.*, 41(1):1–31, 2008.

[20] A. McCreight. *The Mechanized Verification of Garbage Collector Implementations*. PhD thesis, Yale University, New Haven, CT, USA, 2008.

[21] A. McCreight. Practical tactics for separation logic. In *TPHOLs*, volume 5674 of *LNCS*, pp. 343–358. Springer, 2009.

[22] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying GCs and their mutators. In *PLDI*, pp. 468–479. ACM, 2007.

[23] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *TOPLAS*, 21(3):527–568, 1999.

[24] M. O. Myreen. *Formal verification of machine-code programs*. PhD thesis, University of Cambridge, 2008.

[25] W. Partain. The nofib benchmark suite of Haskell programs. In *Proc. 1992 Glasgow Workshop on FP*, pp. 195–202. Springer, 1993.

[26] S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.

[27] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *JFP*, 2(2):127–202, 1992.

[28] S. L. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In *ESOP*, pp. 18–44, 1996.

[29] S. L. Peyton Jones, N. Ramsey, and F. Reig. C–: A portable assembly language that supports garbage collection. In *PPDP '99*, pp. 1–28, London, UK, 1999. Springer-Verlag.

[30] M. Sulzmann, M. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *TLDI*, pp. 53–66, 2007.

[31] The GHC Team. GHC. `http://haskell.org/ghc`, 2009.

[32] The GHC Team. Replacing GMP. `http://hackage.haskell.org/trac/ghc/wiki/ReplacingGMPNotes`, April 2009.

[33] The HASP Project. `http://hasp.cs.pdx.edu`.

[34] The Ocaml Development Team. The Caml language. `http://caml.inria.fr`.

[35] A. Tolmach, T. Chevalier, and the GHC Team. An external representation for the GHC Core language. `http://www.haskell.org/ghc/docs/6.10.4/html/ext-core/core.pdf`, July 2009.

[36] N. Torp-Smith, L. Birkedal, and J. C. Reynolds. Local reasoning about a copying garbage collector. *ACM TOPLAS*, 30(4):1–58, 2008.

[37] J. C. Vanderwaart and K. Crary. A typed interface for garbage collection. In *TLDI*, pp. 109–122. ACM Press, 2003.