Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®
Science of Computer Programming 50 (2004) 101 – 127
Science of Computer Programming
www.elsevier.com/locate/scico

# Building certified libraries for PCC: dynamic storage allocation☆

Dachuan Yu*, Nadeem A. Hamid, Zhong Shao

*Department of Computer Science, Yale University, New Haven, CT 06520-8285, USA*

Received 11 June 2003; received in revised form 12 November 2003; accepted 22 December 2003

**Abstract**

Proof-carrying code (PCC) allows a code producer to provide to a host a program along with its formal safety proof. The proof attests to a certain safety policy enforced by the code, and can be mechanically checked by the host. While this language-based approach to code certification is very general in principle, existing PCC systems have only focused on programs whose safety proofs can be automatically generated. As a result, many low-level system libraries (e.g., memory management) have not yet been handled. In this paper, we explore a complementary approach in which general properties and program correctness are semi-automatically certified. In particular, we introduce a low-level language, CAP, for building certified programs and present a certified library for dynamic storage allocation.
© 2004 Elsevier B.V. All rights reserved.

*Keywords:* Proof-carrying code; Certifying compilation; Dynamic storage allocation; Assembly programming

## 1. Introduction

Proof-carrying code (PCC) is a general framework pioneered by Necula and Lee [15,13]. It allows a code producer to provide a program to a host along with a formal safety proof. The proof is incontrovertible evidence of safety which can be mechanically checked by the host; thus the host can safely execute the program even though the producer may not be trusted.

☆ This research is based on work supported in part by DARPA OASIS grant F30602-99-1-0519, NSF grant CCR-9901011, NSF ITR grant CCR-0081590, and NSF grant CCR-0208618. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.
* Corresponding author.
*E-mail addresses:* yu@cs.yale.edu (D. Yu), hamid-nadeem@cs.yale.edu (N.A. Hamid), shao@cs.yale.edu (Z. Shao).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.01.003

Although the PCC framework is general and potentially applicable to certifying arbitrary data objects with complex specifications [14,2], generating proofs remains difficult. Existing PCC systems [16,12,3,1] have only focused on programs whose safety proofs can be automatically generated. As a result, many low-level system libraries, such as dynamic storage allocation, have not been certified. Nonetheless, building certified libraries, especially low-level system libraries, is an important task in certifying compilation. It not only helps increase the reliability of "infrastructure" software by reusing provably correct program routines, but also is crucial in making PCC scale for production.

On the other hand, Hoare logic [7,8], a widely applied approach in program verification, allows programmers to express their reasonings with assertions and the application of inference rules, and can be used to prove general program correctness. In this paper, we introduce a conceptually simple low-level language for certified assembly programming (CAP) that supports Hoare-logic style reasoning. We use CAP to build a certified library for dynamic storage allocation, and further use this library to build a certified program whose correctness proof can be mechanically checked. Applying Hoare-logic reasoning at an assembly-level, our paper makes the following contributions:

- CAP is based on a common instruction set so that programs can be executed on real machines with little effort. The expected behavior of a program is explicitly written as a specification using higher-order logic. The programmer proves the well-formedness of a program with respect to its specification using logic reasoning, and the result can be checked mechanically by a proof-checker. The soundness of the language guarantees that if a program passes the static proof-checking, its run-time behavior will satisfy the specification.
- Using CAP, we demonstrate how to build certified libraries and programs. The specifications of library routines are precise yet general enough to be imported in various user programs. Proving the correctness of a user program involves linking with the library proofs.
- We implemented CAP and the dynamic storage allocation routines using the Coq proof assistant [20], showing that this library is indeed certified. The example program is also implemented. All the Coq code is available [21].
- Lastly, memory management is an important and error-prone part of most non-trivial programs. It has been a hard problem to address in previously developed frameworks for certified code—most of these assume its correctness and build the rest of the system on top of it. We present a provably correct implementation of a typical dynamic storage allocation algorithm. To the authors' knowledge, it is so far the only such certified library for memory management.

## 2. Dynamic storage allocation

In the remainder of this paper, we focus on the certification and use of a library module for dynamic storage allocation. In particular, we implement a storage allocator similar to that described in [10,11].

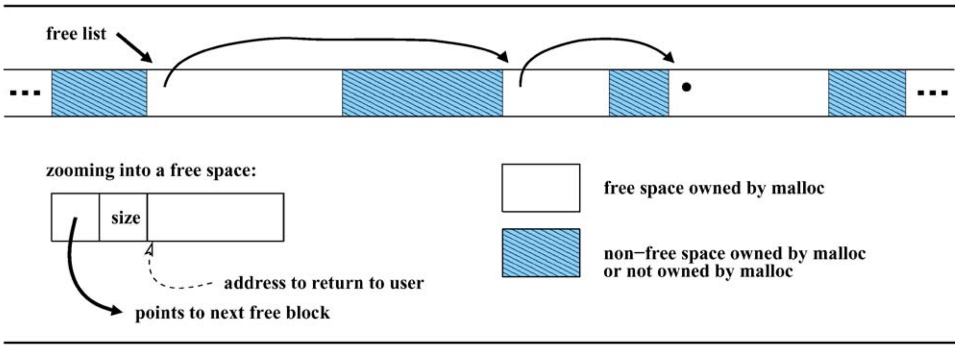
free list
zooming into a free space:
size
address to return to user
points to next free block
free space owned by malloc
non−free space owned by malloc
or not owned by malloc

Fig. 1. Free list and free blocks.

The interface to our allocator consists of the standard malloc and free functions. The implementation keeps track of a *free list* of blocks which are available to satisfy memory allocation requests. As shown in Fig. 1, the free list is a null-terminated list of (non-contiguous) memory blocks. Each block in the list contains a header of two words: the first stores a pointer to the next block in the list, and the second stores the size of the block. The allocated block pointer that is returned to a user program points to the useable space in the block, not to the header (although the header is always carried with the allocated block as well).

The blocks in the list are sorted in order of increasing address and requests for allocation are served based on a first-fit policy; hence, we implement an *address-ordered first-fit* allocation mechanism. If no block in the free list is big enough, or if the free list is empty, then malloc requests more memory from the operating system as needed. When a user program is done with a memory block, it is returned to the free list by calling free, which puts the memory block into the free list at the appropriate position.

Our implementation in this paper is simple enough to understand, yet faithfully represents mechanisms used in traditional implementations of memory allocators [22,10,11]. For ease of presentation, we assume our machine never runs out of memory so malloc will never fail, but otherwise many common low-level mechanisms and techniques used in practice are captured in this example, such as use of a free list, in-place header fields, searching and sorting, and splitting and coalescing (described below). We thus believe our techniques can be as easily applied to a variety of other allocator implementations.

In the remainder of this section, we describe in detail the functionality of the malloc and free library routines (Fig. 2), and give some "pseudo-code" for them. We do not show the calloc (allocate and initialize) and realloc (resize allocated block) routines because they essentially delegate their tasks to the two main functions described below.

**free.** This routine puts a memory block into the free list. It takes a pointer (ptr) to the useable portion of a memory block (preceded by a valid header) and does not return anything. It relies on the preconditions that ptr points to a valid "memory block" and that the free list is currently in a good state (i.e., properly sorted). As shown in

```
void free (void* ptr) {
   hp = ptr - header_size;                               // move to header
   for (prev = nil, p = flist; p <> nil; prev = p, p = p->next)
      if (hp < p) {                                         // found place
         if (hp + hp->size == p)  // join or link with upper neighbor
              hp->size += p->size, hp->next = p->next;
         else hp->next = p;
         if (prev <> nil)         // join or link with lower neighbor
              if (prev + prev->size == hp)
                   prev->size += hp->size, prev->next = hp->next;
              else prev->next = hp;
         else flist = hp;
         return;
      }
   hp->next = nil;              // block's place is at end of the list
   if (prev <> nil)                // join or link with lower neighbor
      if (prev + prev->size == hp)
         prev->size += hp->size, prev->next = hp->next;
      else prev->next = hp;
   else flist = hp;
}
void* malloc (int reqsize) {
   actual_size = reqsize + header_size;
   for(prev = nil, p = flist; ; prev = p, p = p->next)
      if (p==nil) {          // end of free list, request more memory
         more_mem(actual_size);
         prev = nil, p = flist;            // restart the search loop
      } else if (p->size > actual_size + header_size) {
         p->size -= actual_size;    // found block bigger than needed
         p += p->size;              //    by more than a header size,
         p->size = actual_size;     //              so split into two
         return (p + header_size);
      } else if (p->size >= actual_size) { // found good enough block
         if (prev==nil) flist = p->next; else prev->next = p->next;
         return (p + header_size);
      }
}
void more_mem(int req_size) {
  if (req_size < NALLOC) req_size = NALLOC;  // request not too small
  q = alloc(req_size);                         // call system allocator
  q->size = req_size;
  free(q + header_size);                    // put new block on free list
}
```

Fig. 2. Pseudo-code of allocation routines.

Fig. 2, free works by walking down the free list to find the appropriate (address-ordered) position for the block. If the block being freed is directly adjacent with either neighbor in the free list, the two are *coalesced* to form a bigger block.

**malloc.** This routine is the actual storage allocator. It takes the size of the new memory block expected by the user program, and returns a pointer to an available block of memory of that size. As shown in Fig. 2, malloc calculates the actual size of the block needed including the header and then searches the free list for the first available block with size greater than or equal to what is required. If the size of the block found is large enough, it is *split* into two and a pointer to the tail end is returned to the user.

If no block in the free list is large enough to fulfill the request, more memory is requested from the system by calling more_mem. Because this is a comparatively expensive operation, more_mem requests a minimum amount of memory each time to reduce the frequency of these requests. After getting a new chunk of memory from the system, it is appended onto the free list by calling free. The search loop is then restarted because the new chunk of memory might have been coalesced with a previous block during the call to free.

These dynamic storage allocation algorithms often temporarily break certain invariants, which makes it hard to automatically prove their correctness. During intermediate steps of splitting, coalescing, or inserting memory blocks into the free list, the state of the free list or the memory block is not valid for one or more instructions. Thus, a traditional type system would need to be extremely specialized to be able to handle such code.

## 3. A language for certified assembly programming (CAP)

To write our certified libraries, we use a low-level assembly language, CAP, fitted with specifications reminiscent of Hoare-logic. The assertions that we use for verifying the particular dynamic allocation library described in this paper are inspired by Reynolds' "separation logic" [19,18].

The syntax of CAP is given in Fig. 3. A complete program (or, more accurately, machine state) consists of a code heap, a dynamic state component made up of the register file and data heap, and an instruction sequence. The instruction set captures the most basic and common instructions of an assembly language, and includes a primitive alloc command which can be viewed as a system call. The register file is made up of 32 registers and we assume an unbounded heap with integer words of unlimited size for ease of presentation.

Our type system, as it were, is a very general layer of specifications such that assertions can be associated with programs and instruction sequences. Our assertion language (*Assert*) is the calculus of inductive constructions (CiC) [20,17], an extension of the calculus of constructions [4] which is a higher-order typed lambda calculus that corresponds to higher-order predicate logic via the formulae-as-types principle (Curry–Howard isomorphism [9]). In particular, we implement the system described in this paper using the Coq proof assistant [20]. Assertions are thus defined as Coq terms of

$$
\begin{array}{rrl}
(\mathit{Program}) & \mathbb{P} & ::= (\mathbb{C}, \mathbb{S}, \mathbb{I}) \\
(\mathit{CodeHeap}) & \mathbb{C} & ::= \{\mathtt{f} \rightsquigarrow \mathbb{I}\}^* \\
(\mathit{State}) & \mathbb{S} & ::= (\mathbb{H}, \mathbb{R}) \\
(\mathit{Heap}) & \mathbb{H} & ::= \{\mathtt{l} \rightsquigarrow \mathtt{w}\}^* \\
(\mathit{RegFile}) & \mathbb{R} & ::= \{\mathtt{r} \rightsquigarrow \mathtt{w}\}^* \\
(\mathit{Register}) & \mathtt{r} & ::= \{\mathtt{r}_k\}^{k \in \{0 \ldots 31\}} \\
(\mathit{Labels}) & \mathtt{f}, \mathtt{l} & ::= i \ (\mathit{nat\ nums}) \\
(\mathit{WordVal}) & \mathtt{w} & ::= i \ (\mathit{nat\ nums}) \\
(\mathit{InstrSeq}) & \mathbb{I} & ::= \mathsf{c}; \mathbb{I} \mid \mathsf{jd}\ \mathtt{f} \mid \mathsf{jmp}\ \mathtt{r}
\end{array}
\qquad
\begin{array}{rrl}
(\mathit{Command}) & \mathsf{c} & ::= \mathsf{add}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t \\
 & & \mid \mathsf{addi}\ \mathtt{r}_d, \mathtt{r}_s, i \\
 & & \mid \mathsf{sub}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t \\
 & & \mid \mathsf{subi}\ \mathtt{r}_d, \mathtt{r}_s, i \\
 & & \mid \mathsf{mov}\ \mathtt{r}_d, \mathtt{r}_s \mid \mathsf{movi}\ \mathtt{r}_d, i \\
 & & \mid \mathsf{bgt}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f} \\
 & & \mid \mathsf{bgti}\ \mathtt{r}_s, i, \mathtt{f} \\
 & & \mid \mathsf{alloc}\ \mathtt{r}_d[\mathtt{r}_s] \mid \mathsf{ld}\ \mathtt{r}_d, \mathtt{r}_s(i) \\
 & & \mid \mathsf{st}\ \mathtt{r}_d(i), \mathtt{r}_s \\
(\mathit{CdHpSpec}) & \Psi & ::= \{\mathtt{f} \rightsquigarrow \mathsf{a}\}^* \\
(\mathit{Assert}) & \mathsf{a} & ::= \ldots
\end{array}
$$

Fig. 3. Syntax of CAP.

| if $\mathsf{c} =$ | then $\mathtt{AuxStep}(\mathsf{c}, (\mathbb{H}, \mathbb{R})) =$ |
|---|---|
| $\mathsf{add}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t$ | $(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s) + \mathbb{R}(\mathtt{r}_t)\})$ |
| $\mathsf{addi}\ \mathtt{r}_d, \mathtt{r}_s, i$ | $(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s) + i\})$ |
| $\mathsf{sub}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t$ | $(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s) - \mathbb{R}(\mathtt{r}_t)\})$ |
| $\mathsf{subi}\ \mathtt{r}_d, \mathtt{r}_s, i$ | $(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s) - i\})$ |
| $\mathsf{mov}\ \mathtt{r}_d, \mathtt{r}_s$ | $(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s)\})$ |
| $\mathsf{movi}\ \mathtt{r}_d, \mathtt{w}$ | $(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathtt{w}\})$ |

Fig. 4. Auxiliary state update macro.

type State $\rightarrow$ Prop, where the various syntactic categories of the assembly language (such as State) have been encoded using inductive definitions. We give examples of inductively defined assertions used for reasoning about memory in later sections.

### 3.1. Operational semantics

The operational semantics of the assembly language is fairly straightforward and is defined in Figs. 4 and 5. The former figure defines a "macro" relation detailing the effect of simple instructions on the dynamic state of the machine. Control-flow instructions, such as jd (jump direct to a label), jmp (jump to an address in a register), or bgt (conditional branch), do not affect the data heap or register file. The domain of the heap is altered by an alloc command, which increases the domain with a specified number of labels mapped to undefined[1] data. The ld and st commands are used to access or update the value stored at a given label.

Since we intend to model realistic low-level assembly code, we do not have a "halt" instruction. In fact, termination is undesirable since it means the machine has reached

[1] We use _ to indicate an indeterminate value. Note that this feature causes the operational semantics to be non-deterministic.

| $(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{I}) \longmapsto \mathbb{P}$ where | |
|---|---|
| if $\mathbb{I} =$ | then $\mathbb{P} =$ |
| jd $\mathtt{f}$ | $(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{I}')$ where $\mathbb{C}(\mathtt{f}) = \mathbb{I}'$ |
| jmp $\mathtt{r}$ | $(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{I}')$ where $\mathbb{C}(\mathbb{R}(\mathtt{r})) = \mathbb{I}'$ |
| bgt $\mathtt{r}_s, \mathtt{r}_t, \mathtt{f}; \mathbb{I}'$ | $(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{I}')$ when $\mathbb{R}(\mathtt{r}_s) \le \mathbb{R}(\mathtt{r}_t)$; and<br>$(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{I}'')$ when $\mathbb{R}(\mathtt{r}_s) > \mathbb{R}(\mathtt{r}_t)$ where $\mathbb{C}(\mathtt{f}) = \mathbb{I}''$ |
| bgti $\mathtt{r}_s, i, \mathtt{f}; \mathbb{I}'$ | $(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{I}')$ when $\mathbb{R}(\mathtt{r}_s) \le i$; and<br>$(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathbb{I}'')$ when $\mathbb{R}(\mathtt{r}_s) > i$ where $\mathbb{C}(\mathtt{f}) = \mathbb{I}''$ |
| alloc $\mathtt{r}_d[\mathtt{r}_s]; \mathbb{I}'$ | $(\mathbb{C},(\mathbb{H}',\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathtt{l}\}),\mathbb{I}')$<br>where $\mathbb{R}(\mathtt{r}_s) = i$, $\mathbb{H}' = \mathbb{H}\{\mathtt{l} \rightsquigarrow \_, \ldots, \mathtt{l}+i-1 \rightsquigarrow \_\}$<br>and $\{\mathtt{l},\ldots,\mathtt{l}+i-1\} \cap \mathrm{dom}(\mathbb{H}) = \emptyset$ |
| ld $\mathtt{r}_d, \mathtt{r}_s(i); \mathbb{I}'$ | $(\mathbb{C},(\mathbb{H},\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{H}(\mathbb{R}(\mathtt{r}_s)+i)\}),\mathbb{I}')$<br>where $(\mathbb{R}(\mathtt{r}_s)+i) \in \mathrm{dom}(\mathbb{H})$ |
| st $\mathtt{r}_d(i), \mathtt{r}_s; \mathbb{I}'$ | $(\mathbb{C},(\mathbb{H}\{\mathbb{R}(\mathtt{r}_d)+i \rightsquigarrow \mathbb{R}(\mathtt{r}_s)\},\mathbb{R}),\mathbb{I}')$<br>where $(\mathbb{R}(\mathtt{r}_d)+i) \in \mathrm{dom}(\mathbb{H})$ |
| $\mathtt{c}; \mathbb{I}'$ for remaining cases | $(\mathbb{C},\mathtt{AuxStep}(\mathtt{c},(\mathbb{H},\mathbb{R})),\mathbb{I}')$ |

Fig. 5. Operational semantics.

a "stuck" state where, for example, a program is trying to branch to a non-existent code label, or access an invalid data label. We present in the next section a system of inference rules for specifications which allow one to statically prove that a program will never reach such a bad state.

## 3.2. Inference rules

We define a set of inference rules allowing us to prove specification judgments of the following forms:

| | |
|---|---|
| $\Psi \vdash \{\mathsf{a}\}\, \mathbb{P}$ | (well-formed program), |
| $\Psi \vdash \mathbb{C}$ | (well-formed code heap), |
| $\Psi \vdash \{\mathsf{a}\}\, \mathbb{I}$ | (well-formed instruction sequence). |

Programs in our assembly language are written in continuation-passing style because there are no call/return instructions. Hence, we only specify preconditions for instruction sequences (preconditions of the continuations actually serve as the postconditions). If a given state satisfies the precondition, the sequence of instructions will run without reaching a bad state. Furthermore, in order to check code blocks, which are potentially mutually recursive, we require that all labels in the code heap be associated with a precondition—this mapping is our code heap specification, $\Psi$.

*Well-formed code heap and programs.* A code heap is well-formed if the code block associated with every label in the heap is well-formed under the corresponding precondition. Then, a complete program is well-formed if the code heap is well-formed, the current instruction sequence is well-formed under the precondition, and the precondition also holds for the dynamic state.

$$\frac{\Psi = \{\mathtt{f}_1 \leadsto \mathtt{a}_1 \ldots \mathtt{f}_n \leadsto \mathtt{a}_n\} \qquad \Psi \vdash \{\mathtt{a}_i\}\, \mathbb{I}_i \quad \forall i \in \{1 \ldots n\}}{\Psi \vdash \{\mathtt{f}_1 \leadsto \mathbb{I}_1 \ldots \mathtt{f}_n \leadsto \mathbb{I}_n\}} \tag{1}$$

$$\frac{\Psi \vdash \mathbb{C} \qquad \Psi \vdash \{\mathtt{a}\}\, \mathbb{I} \qquad (\mathtt{a}\ \mathbb{S})}{\Psi \vdash \{\mathtt{a}\}\, (\mathbb{C}, \mathbb{S}, \mathbb{I})} \tag{2}$$

It should be noted, as will be formally established at the end of this section, that $\Psi$ is more than the "normal" structural, syntactic code typing environment. The assertions that are mapped to labels in $\Psi$ are fully arbitrary predicates on the state that must be satisfied for the code at that label to be safe to run.

*Well-formed instructions: Pure rules.* The inference rules for instruction sequences can be divided into two categories: pure rules, which do not interact with the data heap, and impure rules, which deal with access and modification of the data heap.

The structure of many of the pure rules is very similar. They involve showing that for all states, if an assertion $\mathtt{a}$ holds, then there exists an assertion $\mathtt{a}'$ which holds on the state resulting from executing the current command and, additionally, the remainder of the instruction sequence is well-formed under $\mathtt{a}'$. We use the auxiliary state update macro defined in Fig. 4 to collapse the rules for arithmetic instructions into a single schema. For control flow instructions, we instead require that if the current assertion $\mathtt{a}$ holds, then the precondition of the label that is being jumped to must also be satisfied.

$$\frac{\begin{array}{l}\mathsf{c} \in \{\mathsf{add}, \mathsf{addi}, \mathsf{sub}, \mathsf{subi}, \mathsf{mov}, \mathsf{movi}\} \\ \forall \mathbb{H}.\forall \mathbb{R}.\, \mathtt{a}\,(\mathbb{H}, \mathbb{R}) \supset \mathtt{a}'\,(\mathtt{AuxStep}(\mathtt{c}, (\mathbb{H}, \mathbb{R}))) \qquad \Psi \vdash \{\mathtt{a}'\}\, \mathbb{I}\end{array}}{\Psi \vdash \{\mathtt{a}\}\, \mathtt{c}; \mathbb{I}} \tag{3}$$

$$\frac{\begin{array}{l}\forall \mathbb{H}.\forall \mathbb{R}.\, (\mathbb{R}(\mathtt{r}_s) \leqslant \mathbb{R}(\mathtt{r}_t)) \supset \mathtt{a}\,(\mathbb{H}, \mathbb{R}) \supset \mathtt{a}'\,(\mathbb{H}, \mathbb{R}) \\ \forall \mathbb{H}.\forall \mathbb{R}.\, (\mathbb{R}(\mathtt{r}_s) > \mathbb{R}(\mathtt{r}_t)) \supset \mathtt{a}\,(\mathbb{H}, \mathbb{R}) \supset \mathtt{a}_1\,(\mathbb{H}, \mathbb{R}) \\ \Psi \vdash \{\mathtt{a}'\}\, \mathbb{I} \qquad \Psi(\mathtt{f}) = \mathtt{a}_1\end{array}}{\Psi \vdash \{\mathtt{a}\}\, \mathsf{bgt}\ \mathtt{r}_s, \mathtt{r}_t, \mathtt{f}; \mathbb{I}} \tag{4}$$

$$\frac{\begin{array}{l}\forall \mathbb{H}.\forall \mathbb{R}.\, (\mathbb{R}(\mathtt{r}_s) \leqslant i) \supset \mathtt{a}\,(\mathbb{H}, \mathbb{R}) \supset \mathtt{a}'\,(\mathbb{H}, \mathbb{R}) \\ \forall \mathbb{H}.\forall \mathbb{R}.\, (\mathbb{R}(\mathtt{r}_s) > i) \supset \mathtt{a}\,(\mathbb{H}, \mathbb{R}) \supset \mathtt{a}_1\,(\mathbb{H}, \mathbb{R}) \\ \Psi \vdash \{\mathtt{a}'\}\, \mathbb{I} \qquad \Psi(\mathtt{f}) = \mathtt{a}_1\end{array}}{\Psi \vdash \{\mathtt{a}\}\, \mathsf{bgti}\ \mathtt{r}_s, i, \mathtt{f}; \mathbb{I}} \tag{5}$$

$$\frac{\forall \mathbb{S}.\, \mathtt{a}\ \mathbb{S} \supset \mathtt{a}_1\ \mathbb{S} \quad \text{where } \Psi(\mathtt{f}) = \mathtt{a}_1}{\Psi \vdash \{\mathtt{a}\}\, \mathsf{jd}\ \mathtt{f}} \tag{6}$$

$$\frac{\forall \mathbb{H}.\forall \mathbb{R}.\, \mathtt{a}\,(\mathbb{H}, \mathbb{R}) \supset \mathtt{a}_1\,(\mathbb{H}, \mathbb{R}) \quad \text{where } \Psi(\mathbb{R}(\mathtt{r})) = \mathtt{a}_1}{\Psi \vdash \{\mathtt{a}\}\, \mathsf{jmp}\ \mathtt{r}} \tag{7}$$

*Well-formed instructions: Impure rules.* As mentioned previously, these rules involve accessing or modifying the data heap.

$$\frac{\begin{array}{l}\forall \mathbb{H}.\forall \mathbb{R}.\mathtt{a}\,(\mathbb{H}, \mathbb{R}) \supset \mathtt{a}'(\mathbb{H}\{1 \leadsto \_, \ldots, 1 + i - 1 \leadsto \_\}, \mathbb{R}\{\mathtt{r}_d \leadsto 1\}) \\ \text{where } \mathbb{R}(\mathtt{r}_s) = i \text{ and } \{1, \ldots, 1 + i - 1\} \cap \mathrm{dom}(\mathbb{H}) = \emptyset \\ \Psi \vdash \{\mathtt{a}'\}\mathbb{I}\end{array}}{\Psi \vdash \{\mathtt{a}\}\mathtt{alloc}\ \mathtt{r}_d[\mathsf{s}]; \mathbb{I}} \tag{8}$$

$$\frac{\begin{array}{l}\forall \mathbb{H}.\forall \mathbb{R}.\mathtt{a}\,(\mathbb{H},\mathbb{R} \supset ((\mathbb{R}(\mathtt{r}_s)+i) \in \mathrm{dom}(\mathbb{H})) \wedge (\mathtt{a}'\,(\mathbb{H},\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{H}(\mathbb{R}(\mathtt{r}_s)+i\})) \\ \Psi \vdash \{\mathtt{a}'\}\mathbb{I}\end{array}}{\Psi \vdash \{\mathtt{a}\}\mathsf{ld}\,\mathtt{r}_d,\mathtt{r}_s(i);\,\mathbb{I}} \tag{9}$$

$$\frac{\begin{array}{l}\forall \mathbb{H}.\forall \mathbb{R}.\mathtt{a}\,(\mathbb{H},\mathbb{R} \supset ((\mathbb{R}(\mathtt{r}_d)+i) \in \mathrm{dom}(\mathbb{H})) \wedge (\mathtt{a}'\,(\mathbb{H},\mathbb{R}\{\mathtt{r}_d+i \rightsquigarrow \mathbb{R}(\mathtt{r}_s)\},\mathbb{R})) \\ \Psi \vdash \{\mathtt{a}'\}\mathbb{I}\end{array}}{\Psi \vdash \{\mathtt{a}\}\mathsf{st}\,\mathtt{r}_d,\mathtt{r}_s(i);\,\mathbb{I}} \tag{10}$$

### 3.3. Soundness

We establish the soundness of these inference rules with respect to the operational semantics of the machine following the syntactic approach of proving type soundness [23]. From "Type Preservation" and "Progress" lemmas (proved by induction on $\mathbb{I}$), we can guarantee that given a well-formed program, the current instruction sequence will be able to execute without getting "stuck." Furthermore, at the point when the current instruction sequence branches to another code block, the machine state will always satisfy the precondition of that block.

**Lemma 1** (Type Preservation). *If* $\Psi \vdash \{\mathtt{a}\}(\mathbb{C},\mathbb{S},\mathbb{I})$ *and* $(\mathbb{C},\mathbb{S},\mathbb{I})\longmapsto\mathbb{P}$, *then there exists an assertion* $\mathtt{a}'$ *such that* $\Psi \vdash \{\mathtt{a}'\}\,\mathbb{P}$.

**Lemma 2** (Progress). *If* $\Psi \vdash \{\mathtt{a}\}(\mathbb{C},\mathbb{S},\mathbb{I})$, *then there exists a program* $\mathbb{P}$ *such that* $(\mathbb{C},\mathbb{S},\mathbb{I})\longmapsto\mathbb{P}$.

**Theorem 1** (Soundness). *If* $\Psi \vdash \{\mathtt{a}\}(\mathbb{C},\mathbb{S},\mathbb{I})$, *then for all natural number n, there exists a program* $\mathbb{P}$ *such that* $(\mathbb{C},\mathbb{S},\mathbb{I})\longmapsto^{n}\mathbb{P}$, *and*
- *if* $(\mathbb{C},\mathbb{S},\mathbb{I})\longmapsto^{*}(\mathbb{C},\mathbb{S}',\mathsf{jd}\ \mathtt{f})$, *then* $\Psi(\mathtt{f})\ \mathbb{S}'$;
- *if* $(\mathbb{C},\mathbb{S},\mathbb{I})\longmapsto^{*}(\mathbb{C},(\mathbb{H},\mathbb{R}),\mathsf{jmp}\ \mathtt{r}_d)$, *then* $\Psi(\mathbb{R}(\mathtt{r}_d))\ (\mathbb{H},\mathbb{R})$;
- *if* $(\mathbb{C},\mathbb{S},\mathbb{I})\longmapsto^{*}(\mathbb{C},(\mathbb{H},\mathbb{R}),(\mathsf{bgt}\ \mathtt{r}_s,\mathtt{r}_t,\mathtt{f}))$ *and* $\mathbb{R}(\mathtt{r}_s) > \mathbb{R}(\mathtt{r}_t)$, *then* $\Psi(\mathtt{f})\ (\mathbb{H},\mathbb{R})$;
- *if* $(\mathbb{C},\mathbb{S},\mathbb{I})\longmapsto^{*}(\mathbb{C},(\mathbb{H},\mathbb{R}),(\mathsf{bgti}\ \mathtt{r}_s,i,\mathtt{f}))$ *and* $\mathbb{R}(\mathtt{r}_s) > i$, *then* $\Psi(\mathtt{f})\ (\mathbb{H},\mathbb{R})$.

It should be noted here that this soundness theorem establishes more than simple type safety. In addition to that, it states that whenever we jump to a block of code in the heap, the specified precondition of that code (which is an arbitrary assertion) will hold.

It may seem that the inference rules given above for instructions will be hard to apply since they require the introduction of assertions which are not mentioned in the syntax of the program. Of course, these assertions must be produced somehow, either automatically or by the programmer. Additionally, it would be possible to extend the definition of the syntax of CAP so that every instruction in a sequence is annotated with its precondition. Such a user-friendly, but verbose, syntax has been used in the presentation of CAP programs in the Appendix. Ultimately, however, recall that we will be using the CAP system to convey PCC packages. In this scenario, what the code recipient gets from the producer will be a complete proof of the well-formedness of a program. The proof will consist of a tree of applications of the appropriate CAP inference rules and embedded therein will be any necessary information, especially preconditions at the intermediate steps of a code block. To keep the presentation less cluttered, we have defined the syntax of CAP to only provide assertions at the entry points of code blocks; the code producer (a human or an automatic process like a certifying compiler) will somehow have to keep track of any intermediate assertions, and all these will eventually be packaged in the complete proof that is provided to the code recipient.

## 4. Certified dynamic storage allocation

Equipped with CAP, we are ready to build the certified library. In particular, we provide a provably correct implementation for the library routines free and malloc. The main difficulties involved in this task are: (1) to give precise yet general specifications to the routines; (2) to prove as theorems the correctness of the routines with respect to their specifications; (3) the specifications and theorems have to be modular so that they can interface with user programs. In this section, we discuss these problems for free and malloc respectively. From now on, we use the word "specification" in the wider sense, meaning anything that describes the behavior of a program. To avoid confusion, we call the language construct $\Psi$ a *code heap spec*, or simply *spec*.

Before diving into certifying the library, we define some assertions related to memory blocks and the free list as shown in Fig. 6. These definitions make use of some basic operators (which we implement as shorthands using primitive constructs) commonly seen in separation logic [19,18]. In particular, **emp** asserts that the heap is empty; $e \mapsto e'$ asserts that the heap contains one cell at address $e$ which contains $e'$; and separating conjunction $p{*}q$ asserts that the heap can be split into two disjoint parts in which $p$ and $q$ hold, respectively.

Memory block ($\mathsf{MBlk}\,p\,q\,s$) asserts that the memory at address $p$ is preceded by a pair of words: the first word contains $q$, a (possibly null) pointer to another memory block, and the second word contains the size, $s$, of the memory block itself (including the two-word header preceding $p$).

Memory block list ($\mathsf{MBlkLst}\,n\,p\,q$) models an address-ordered list of blocks. $n$ is the number of blocks in the list, $p$ is the starting pointer and $q$ is the ending pointer. This assertion is defined inductively and is a specialized version of the *singly linked list* introduced by Reynolds [19,18]. However, unlike that somewhat informal definition of singly linked list, MBlkLst has to be defined formally for mechanical proof-checking. Thus we use a Coq inductive definition for this purpose.

A list segment with a particular ending block ($\mathsf{EndL}\,\mathit{flist}\,pq$) is defined as a list *flist* of memory blocks with $p$ pointing at the last block whose forward pointer is $q$. In the special case that *flist* is an empty list, $p$ is `nil`.
($\mathsf{MidL}\,\mathit{flist}\,p\,q$) models a list with a block $B$ in the middle, where the list starts from *flist*, and the block $B$ is specified by the position $p$ and the forward pointer $q$. This assertion is defined as the separating conjunction of a list with ending block $B$ and a null-terminated list starting from the forward pointer of $B$.

$$\begin{aligned}
&\mathsf{MBlk}\ p\ q\ s\\
&\quad\equiv (p > 2) \wedge (s > 2) \wedge\\
&\qquad (p-2 \mapsto q) * (p-1 \mapsto s)\\
&\qquad * (p, \ldots, p+s-3 \mapsto \_, \ldots, \_)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{MBlkLst}\ 0\ p\ q\\
&\quad\equiv \mathbf{emp} \wedge (p = q)\\
&\mathsf{MBlkLst}\ (n+1)\ p\ q\\
&\quad\equiv \exists p'.(\mathsf{MBlk}\ (p+2)\ p'\ \_)\\
&\qquad\qquad *(\mathsf{MBlkLst}\ n\ p'\ q)\\
&\qquad\qquad \wedge(p < p' \vee p' = \mathtt{nil})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{EndL}\ \mathit{flist}\ p\ q\\
&\quad\equiv ((p = \mathtt{nil}) \supset (\mathsf{MBlkLst}\ 0\ \mathit{flist}\ q))\\
&\qquad \wedge(p \neq \mathtt{nil}\\
&\qquad\quad \supset \exists n.\,((\mathsf{MBlkLst}\ n\ \mathit{flist}\ p)\\
&\qquad\qquad\qquad *(\mathsf{MBlk}\ (p+2)\ q\ \_)\\
&\qquad\qquad\qquad \wedge(p < q \vee q = \mathtt{nil})))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{MidL}\ \mathit{flist}\ p\ q\\
&\quad\equiv \exists n.(\mathsf{EndL}\ \mathit{flist}\ p\ q)\\
&\qquad\qquad *(\mathsf{MBlkLst}\ n\ q\ \mathtt{nil})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{Good}\ \mathit{flist}\\
&\quad\equiv \exists n.(\mathsf{MBlkLst}\ n\ \mathit{flist}\ \mathtt{nil})
\end{aligned}$$

$$\forall p.\forall q.\,(\mathsf{MidL}\ \mathit{flist}\ p\ q) \supset (\mathsf{Good}\ \mathit{flist})$$

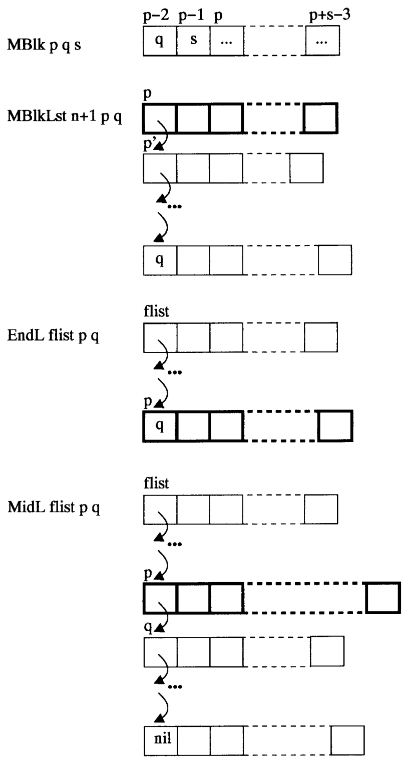

Fig. 6. Assertions on free list.

Finally, we define a good free list (Good) as a null-terminated memory block list. It is easy to show the relation between MidL and Good as described.

**free.** Putting aside the CAP syntax for the moment, a specification of the expected behavior of free can be written as the following Hoare triple:

$$\begin{aligned}
&\{\mathit{PRE}\}\ \mathsf{free}(\mathit{fptr})\ \{\mathit{POST}\};\\
&\text{where } \mathit{PRE} \equiv \mathit{Pred} * (\mathsf{MBlk}\ \mathit{fptr}\ \_\ \_) * (\mathsf{Good}\ \mathit{flist})\\
&\qquad\quad \mathit{POST} \equiv \mathit{Pred} * (\mathsf{Good}\ \mathit{flist})
\end{aligned}$$

Assertion *PRE* states the precondition. It requires that the heap can be separated into three disjoint parts, where *fptr* points to a memory block to be freed; *flist* points to a good free list; and the remaining part satisfies the user specified assertion *Pred*. Assertion *POST* states the postcondition. Since the memory block is placed into the free list, the heap now can be separated into two disjoint parts: *flist* still points to a good free list, and the remaining part of the heap still satisfies *Pred* because it is untouched.

Note that this does not totally specify all the behaviors of free. For example, it is possible to add in the postcondition that the memory block that *fptr* pointed to is now in the free list. However, this is irrelevant from a library user's point of view. Thus we favor the above specification, which guarantees that free does not affect the remaining part of the heap.

Now we write this specification in CAP, where programs are written in continuation-passing style. Before free completes its job and jumps to the return pointer, the postcondition should be established. Thus the postcondition can be interpreted as the precondition of the code referred to by the return pointer. Suppose $\mathtt{r}_0$ is the return pointer; a valid library call to free should require that *POST* implies $\Psi(\mathbb{R}(\mathtt{r}_0))$ for all states (which we write as $POST \Longrightarrow \Psi(\mathbb{R}(\mathtt{r}_0))$). In fact, this condition is required for type-checking the returning code of free (i.e., jmp $\mathtt{r}_0$). As a library routine, free is expected to be used in various programs with different code heap specs ($\Psi$). So the above condition has to be established by the user with their knowledge of the actual, complete $\Psi$. When proving the well-formedness of free, this condition is taken as a premise.

At an assembly-level, most non-trivial programs are expressed as multiple code blocks connected together with control flow instructions (jd, jmp, and bgt). The set of code blocks implementing the free routine is given in Fig. 11. Type-checking these control flow instructions requires similar knowledge about the code heap spec $\Psi$. For instance, at the end of the initial code block `free`,[2] an assertion $A_{iter}$ (i.e., the precondition of jd `iter`) is established about the current state, and the control is transferred to the code block `iter` with a direct jump. When type-checking this direct jump (i.e., jd `iter`) against the assertion $A_{iter}$, the inference rule 6 requires that $A_{iter}$ implies $\Psi(\mathtt{iter})$ for all states. These requirements are also taken as premises in the well-formedness theorem of `free`. Thus the specification of `free` is actually as follows:

$$\begin{aligned}\forall Pred.\forall\Psi.\forall\mathtt{f}.\,(POST \Longrightarrow \Psi(\mathtt{f})) \wedge (A_{iter} \Longrightarrow \Psi(\mathtt{iter}))\\ \supset \Psi \vdash \{PRE \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}\}\, \mathbb{C}(\mathtt{free})\end{aligned}$$

where $\mathbb{C}(\mathtt{free})$ is the code block labeled `free`, $\mathtt{r}_0$ holds the return location, and universally quantified *Pred* occurs inside the macros *PRE* and *POST* as defined before. This is defined as a theorem and formally proved in Coq.

The first hypothesis, $POST \Longrightarrow \Psi(\mathtt{f})$, ensures that the entire free routine can eventually be exited by jumping to $\mathtt{r}_0$; the second establishes that the initial `free` block can be left by jumping to `iter`.

Following similar ideas, the well-formedness of all the other code blocks implementing the library routine free are also modeled and proved as theorems, with the premises changed appropriately according to which labels they refer to. See Section 5.1 and the appendix for details.

Using the Coq proof assistant, proving these theorems is not difficult. Pure instructions only affect the register file; they are relatively easy to handle. Impure instructions affect the heap. Nonetheless, commonalities on similar operations can be factored out as lemmas. For instance, writing into the "link" field of a memory block header occurs in various places. By factoring out this behavior as a lemma and applying it, the proof construction becomes simple routine work. The only tricky part lies in proving the

[2] Note we use a different font to distinguish between free, which is the collection of code blocks implementing the "free" procedure, and `free`, which is the individual code block labeled "free."

code which performs coalescing of free blocks. This operation essentially consists of two steps: one to modify the size field; the other to combine the blocks. No matter which one is performed first, one of the blocks has to be "broken" from being a valid memory block as required by MBlk. This behavior is hard to handle in conventional type systems, because it tends to break certain invariants captured by the type system. In contrast, since the predicates of CAP are expressed using the complete generality of the higher-order logic, the intermediate steps of an operation can be verified, for example, by expressing in the assertions exactly how the blocks are being broken up and put back together again to form a larger (valid) block. In our case, we used an abstraction inspired by separation logic to allow more intuitive reasoning about these low-level memory operations.

In Fig. 11 of Appendix A, we give the routine free written in CAP. This program is annotated with assertions at various program points. It contains the spec templates (the assertions at the beginning of every code block), and can be viewed as an outline of the proof. In this program, variables are used instead of register names for ease of understanding. We also assume all registers to be caller-saved, so that updating the register file does not affect the user customized assertion *Pred*. Typically, relevant states are saved in activation records in a stack when making function calls, and *Pred* would be dependent only on the stack. In the current implementation, we have not yet provided certified activation records; instead, we simply use different registers for different programs. (For a more complete implementation, we would need a more robust way of interfacing between the library and user code—using a stack, calling conventions, etc.)

A certified library routine consists of both the code and the proof. Accordingly, the interface of such a routine consists of both the signature (parameters) and the spec templates (e.g., *PRE*, *POST*). When the routine is used by a user program, both the parameters and the spec templates should be instantiated properly. The well-formedness of `free` is also a template which can be applied to various assertions *Pred*, code heap specs $\Psi$ and return labels $f$. If a user program contains only one call-site to `free`, the corresponding assertion for `free` should be used in $\Psi$. However, if a user program contains multiple call-sites to `free`, a "sufficiently weak" assertion for `free` must be constructed by building a "tagged" disjunction of all the individually instantiated assertions.[3] The following derived Rule 11 (which is proved by induction on $\mathbb{I}$), together with the theorem for the well-formedness of `free`, guarantees that the program type-checks.

$$\frac{\Psi \vdash \{\mathsf{a}_1\}\,\mathbb{I} \qquad \Psi \vdash \{\mathsf{a}_2\}\,\mathbb{I}}{\Psi \vdash \{\mathsf{a}_1 \vee \mathsf{a}_2\}\,\mathbb{I}} \tag{11}$$

**malloc.** Just as for free, an informal specification of malloc can be as follows:

$$\begin{aligned} &\{PRE\}\ \mathsf{malloc}(nsize, mptr)\ \{POST\};\\ &\text{where } PRE \equiv Pred * (\mathsf{Good}\ flist) \wedge (nsize = s_0 > 0)\\ &\qquad\quad POST \equiv Pred' * (\mathsf{Good}\ flist) * (\mathsf{MBlk}\ mptr\ \_\ s) \wedge\ (s_0 + 2 \leqslant s)\end{aligned}$$

[3] See the discussion in Section 5.1.

The precondition *PRE* states that *flist* points to a good free list, user customized assertion *Pred* holds for the remaining part of the heap, and the requested size *nsize* is larger than 0. The postcondition *POST* states that part of the heap is the newly allocated memory block pointed to by *mptr* whose size is at least the requested one, *flist* still points to a good free list, and another assertion *Pred′* holds for the remaining part of the heap. *Pred′* may be different from *Pred* because malloc modifies register *mptr*. The relation between these two assertions is described by *SIDE* as follows:

$$SIDE \equiv \forall(\mathbb{H},\mathbb{R}).Pred\,(\mathbb{H},\,\mathbb{R}) \supset Pred'\,(\mathbb{H},\mathbb{R}\{mptr \rightsquigarrow \_\})$$

Typically, *Pred* does not depend on *mptr*. So *Pred′* is the same as *Pred* and the above condition is trivially established.

To type-check the control-flow instructions of the malloc routine without knowing the actual code heap spec $\Psi$, we add premises to the well-formedness theorem of `malloc` just as we did for `free`. The specification in CAP is as follows:

$$\begin{aligned}\forall Pred.\forall Pred'.\forall s_0.\forall\Psi.\forall \mathtt{f}.\,SIDE \wedge (POST \Longrightarrow \Psi(\mathtt{f})) \wedge (A_{init} \Longrightarrow \Psi(\mathtt{init}))\\ \supset \Psi \vdash \{PRE \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}\,\mathbb{C}(\mathtt{malloc})\end{aligned}$$

where $\mathbb{C}$(`malloc`) is the code block labeled `malloc`, universally quantified *Pred*, *Pred′* and $s_0$ occur inside the macros *PRE*, *POST* and *SIDE*, `init` is the label of a code block that `malloc` refers to, and $A_{init}$ is the assertion established when `malloc` jumps to `init`. Because `malloc` calls `free` in the course of its execution, we use a different register $\mathtt{r}_1$ to hold the return location for the malloc routine, due to the lack of certified activation records. The well-formedness of all the other code blocks implementing the malloc routine are modeled similarly.

Proving these theorems is not much different than proving those of `free`. A tricky part is the splitting of memory blocks. Similar to coalescing, splitting temporarily breaks certain invariants; thus it is hard to handle in conventional type systems. The annotated malloc routine in CAP is shown in Fig. 12.

## 5. Example: copy program

With the certified implementation (i.e., code and proof) of free and malloc, we now implement a certified program, copy. As shown in Fig. 7, this copy program takes a pointer to a list as the argument, makes a copy of the list, and disposes the original one.

To make use of the certified routines free and malloc, we define assertions for the list data structure as shown in Fig. 8. ($\mathsf{Pair}\,p\,x\,q$) defines a pair at location $p$ which stores values $x$ and $q$; it carries the fact that it resides inside a “malloced” memory block. ($\mathsf{Slist}\,\alpha\,p\,q$) defines a list with the help of Pair; it represents a list segment from $p$ to $q$ representing the sequence $\alpha$. The structure of the Slist definition is close to that of MBlkLst and Reynolds’ *singly-linked list* [19,18].

The MBlk assertion carried inside Pair is crucial for the memory block to be “freed” when required. It has to be preserved throughout the copy program. Typically when operating on a pair at location $p$, only locations $p$ and $p+1$ are referred to. Thus as

```
list* copy (list* src) {
   target = prev = nil;
   while (src<>nil) {
      p = malloc(2);                       \\ allocate for a new element
      p->data = src->data, p->link = src->link;    \\ copy an element
      old = src, src = src->link, free(old);       \\ dispose old one
      if (prev == nil) {target = p, prev = p}
      else {prev->link = p, prev=p}            \\ link in new element
   }
   return target;
}
```

Fig. 7. Pseudo-code of copy.

$$\begin{aligned}
&\mathsf{Pair}\; p\; x\; q \equiv \forall(\mathbb{H},\mathbb{R}).\,\exists lnk.\,\exists siz.\,(\mathbb{H}(p)=x)\wedge(\mathbb{H}(p+1)=q)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge(\mathsf{MBlk}\; p\; lnk\; siz)\wedge(siz-2\geq 2)\\
&\mathsf{Slist}\;\epsilon\; p\; q \equiv \mathbf{emp}\wedge(p=q)\\
&\mathsf{Slist}\;(x\cdot\alpha)\; p\; q \equiv \exists p'.(\mathsf{Pair}\; p\; x\; p')*(\mathsf{Slist}\;\alpha\; p'\; q)
\end{aligned}$$

Fig. 8. Pair and Slist.

long as the header of the memory block is untouched, preserving MBlk is straightforward.

Certifying the copy program involves the following steps: (1) write the plain code; (2) write the code heap spec; (3) prove the well-formedness of the code with respect to the spec, with the help of the library proofs. Fig. 13 of the Appendix shows the copy program with annotations.

The spec for the code blocks that implement the copy program depends on what property one wants to achieve. In our example, we specify the partial correctness that if copy ever completes its task (by jumping to `halt`), the result list contains the same sequence as the original one.

We get the specs of the library blocks by instantiating the spec templates of the previous section with appropriate assertion *Pred*. The only place where `malloc` is called is in block `nxt0` of copy. Inspecting the assertion at that place and the spec template, we instantiate *Pred* appropriately to get the actual spec. Although `free` is called only once in program copy (in block `nxt1`), it has another call-site in block `more` of `malloc`. Thus for any block of `free`, there are two instantiated specs, one customized for copy ($A_1$) and the other for malloc ($A_2$). The actual spec we use is the disjunction of these two ($A_1 \vee A_2$).

The well-formedness of the program can be derived from the well-formedness of all the code blocks. We follow the proof outline in Fig. 13 to handle the blocks of copy. For the blocks of routine malloc, we directly import their well-formedness theorems described in the previous section. Proving the premises of these theorems (e.g., $A_{init} \Longrightarrow \Psi(\mathtt{init})$) is trivial (e.g., $A_{init}$ is exactly $\Psi(\mathtt{init})$). For routine free, whose spec has a disjunctive form, we apply Rule 11 to break up the disjunction and apply the theorems twice. Proving the premises of these theorems (e.g., $A_{iter} \Longrightarrow \Psi(\mathtt{iter})$ where $\Psi(\mathtt{iter})$ has the form $A_{iter} \vee A'_{iter}$) involves $\vee$-*introduction*, which is also trivial. We refer interested readers to our implementation [21] for the exact details.

## 5.1. Higher-order code pointers

We have previously mentioned several times the need for instantiating specifications with a disjunction of assertions. This arises from a deficiency in Hoare logic's handling of higher-order, or embedded, code pointers. It is an issue that has also been mentioned in [19] and which has been inherited by the version of the CAP language presented in this paper. A simple outline of the problem is as follows: We wish to specify a precondition for free, for example, which says that there is some return code address ($g$) in register $\mathtt{r}_0$, that $g$ has its own particular assertion about the state of the memory and register file ($A_g$), and that at the point when the code of free is entered, $A_g$ and some other separate assertion, $A_{\mathsf{free}}$, must hold.[4] Thus, we would like to express the specification for the `free` block as follows:

$$\Psi(\mathtt{free}) = \{\exists g.\, \mathbb{R}(\mathtt{r}_0) = g \wedge \Psi(g) = A_g \wedge A_g * A_{\mathsf{free}}\}.$$

Unfortunately, the reader will note that this results in a circular definition of $\Psi$. The approach taken to circumvent this in this paper is, as mentioned previously, to explicitly list all possible return code addresses and their preconditions in the specification of free. Thus, in the case of the copy program, by analyzing the call paths to free, we see that when the `free` block is entered, the return code address will either be `nxt0` or `nxt1`, and each of those blocks has its own associated assertion, $A_{\mathtt{ntx0}}$ and $A_{\mathtt{ntx1}}$. Thus, we give the `free` block the specification:

$$\Psi(\mathtt{free}) = \{((\mathbb{R}(\mathtt{r}_0) = \mathtt{nxt0} \wedge A_{\mathtt{nxt0}}) \vee (\mathbb{R}(\mathtt{r}_0) = \mathtt{nxt1} \wedge A_{\mathtt{ntx1}})) * A_{\mathtt{free}}\}.$$

This is what is meant by instantiating *Pred* with a disjunction of assertions, as discussed in the previous subsection. Now, each particular code block of the free routine has its own specification about the state that is needed to prove it safe, just as `free` has $A_{\mathsf{free}}$. The precondition of each code block, then, is a combination of the disjunction related to the return code addresses and the specific assertion about the memory and register file that is needed by that code block to reason safety of its operations. While jumping between the code blocks implementing the free routine, we can keep the disjunctive assertions consistent by checking the value of the return address in $\mathtt{r}_0$.

Clearly, this method of handling return code addresses is far from satisfactory. Although it is somewhat alleviated by introducing the derived Rule 11, we have already

[4] Here, $A_g$ and $A_{\mathsf{free}}$ correspond to *Pred* and $((\mathsf{MBlk}\ fptr\_\_) * (\mathsf{Good}\ flist))$, respectively, in the precondition, *PRE*, of free described in Section 4.

worked on developing the next generation of CAP, which more properly addresses this problem.

## 6. Implementation

### 6.1. Overview

Before presenting the implementation, we first clarify and emphasize some important points. Firstly, the goal of this paper has been to present a framework for proof-carrying code. In the context of the most foundational form of PCC (e.g., [1]), the framework basically includes only the actual machine (which we use an idealized one in this presentation) and a logic in which proofs about the safety of the machine state and its operation are given. The logic we are using is CiC (Coq), as mentioned previously. Thus, the presentation of the CAP language in Fig. 3 is somewhat inverted, since the elided definition of *Assert* (which is the complete syntax of Coq) is really the only syntax of our system. All the other parts of that figure are actually defined as Coq terms as will be detailed further in this section. Since the entire CAP language is actually embedded in Coq, we have the use of Coq's full expressiveness, including its facility for inductive definitions, as shown especially in the beginning of Section 4.

Furthermore, the operational semantics and typing rules presented in Section 3 are also defined as a collection of relations in Coq. Ultimately, they allow one to formally prove the CAP soundness theorem, which ensures the stated properties of the machine's execution, given a Coq term that represents a well-formed CAP program.

Our implementation [21] in Coq covers the language CAP and its soundness, the certified routines `free` and `malloc`, and the example program `copy`. The code and initial state of the library routines and program are Coq terms with appropriate types. Their well-formedness is a Coq theorem which is constructed interactively using tactics with the help of the Coq proof assistant.

### 6.2. Syntax

Most of the definitions for CAP syntax are fairly intuitive. For better correspondence with the presentation in Fig. 3, we present the definitions top-down. In the actual Coq implementation, they are defined bottom-up so that definitions only refer to others which are already defined.

A program is a triple consisting of a code heap, a dynamic state and an instruction sequence. The code heap is inductively defined; it exhibits a list structure. Although the definition itself does not prevent the same label from being mapped to multiple targets, the lookup relation `lookupC`, which corresponds to the judgment "$\mathbb{C}(\mathtt{f}) = \mathbb{I}$", makes sure that only the right-most one is visible.

```
Definition prog := (codeheap * (state * iseq)).

(* code labels are defined as natural numbers *)
Definition lab := nat.
```

```
Inductive codeheap : Set :=
 | emptyC : codeheap
 | consC  : lab -> iseq -> codeheap -> codeheap.

Inductive lookupC : codeheap -> lab -> iseq -> Prop := ...
```

The dynamic state is made up of the data heap and the register file. The heap is defined as a partial function from heap labels to word values. In this encoding, if a heap maps a label to `none`, it indicates that the label is not in the domain of the heap. A library of utility functions and lemmas is built to help manipulate heaps and proving their properties in this encoding. The register file is inductively defined; its structure is similar to that of the code heap. Registers are defined as an inductive set with 32 constructors—one for each register that we have in CAP. The function `lookupR` looks up a value in the register file.

```
Definition state := (heap * rfile).

(* heap labels and word values are defined as natural numbers *)
Definition addr    := nat.
Definition wordval := nat.

Definition heap : Set := (addr -> (option wordval)).

Inductive rfile : Set :=
 | emptyR : rfile
 | consR  : rfile -> reg -> wordval -> rfile.

Inductive reg : Set := r0 : reg | r1 : reg | r2 : reg | ...

Definition lookupR : rfile -> reg -> int := ...
```

The definitions of instruction sequences and commands are straightforward. Code heap types are defined similarly as code heaps, except that the range of the mapping is assertions (which are simply predicates on states).

```
Inductive iseq : Set :=
 | seq : comm -> iseq -> iseq
 | jd  : lab -> iseq
 | jmp : reg -> iseq.

Inductive comm : Set :=
 | add   : reg -> reg -> reg -> comm
 | addi  : reg -> reg -> nat -> comm
 | bgt   : reg -> reg -> lab -> comm
 | alloc : reg -> reg -> comm
 | ld    : reg -> reg -> nat -> comm
 | st    : reg -> nat -> reg -> comm
 | ...

Definition assert := state -> Prop.
```

```
Inductive Step : prog -> prog -> Prop :=
 | ev_mov  : (ch:codeheap; H:heap; R:rfile; rd,rs:reg; b:iseq)
             (Step (ch, ((H, R), (seq (mov rd rs) b)))
                   (ch, ((H, (updateR R rd (lookupR R rs))), b)))
 | ev_bgtF : (ch:codeheap; H:heap; R:rfile; rs,rt:reg; f:lab; b:iseq)
             (le (lookupR R rs) (lookupR R rt))    (* rs <= rt *)
             -> (Step
                   (ch, ((H,R), (seq (bgt rs rt f) b)))
                   (ch, ((H,R), b)))
 | ev_bgtT : (ch:codeheap;H:heap;R:rfile;rs,rt:reg;f:lab;b,b':iseq)
             (gt (lookupR R rs) (lookupR R rt))    (* rs > rt *)
             -> (lookupC ch f b')
             -> (Step
                   (ch, ((H,R), (seq (bgt rs rt f) b)))
                   (ch, ((H,R), b')))
 | ev_jd   : (ch:codeheap; s:state; f:lab; b:iseq)
             (lookupC ch f b)
             -> (Step (ch, (s, (jd f))) (ch, (s, b)))
 | ev_jmp : (ch:codeheap; H:heap; R:rfile; r:reg; b:iseq)
            (lookupC ch (lookupR R r) b)
            -> (Step (ch, ((H,R), (jmp r))) (ch, ((H,R), b)))
 | ...
```

Fig. 9. Coq encoding of CAP operational semantics.

```
Inductive codeheapty : Type :=
 | emptyCT : codeheapty
 | consCT  : lab -> assert -> codeheapty -> codeheapty.

Inductive lookupCT : codeheapty -> lab -> assert -> Prop := ...
```

### 6.3. Operational semantics, inference rules and soundness

The operational semantics is encoded as a relation between programs. Every case of the operational semantics corresponds to a constructor of the inductively defined Step relation. Representative cases of the Step definition are shown in Fig. 9, where lookupR and updateR are utility functions defined to look up and update the register files respectively.

Each judgment form of the inference rules is encoded as a relation using an inductive definition. Part of these definitions are shown in Fig. 10.

The soundness of CAP trivially follows from the “Progress” and “Type Preservation” lemmas. The proofs of these lemmas are straightforward by induction on the instruction sequence I.

```
(* well-formed instruction sequence *)
Inductive Infer : codeheapty -> assert -> iseq -> Prop :=
 | I_mov : (ct:codeheapty; a,a':assert; rd,rs:reg; b:iseq)
           ((H:heap; R:rfile)
            (a (H, R)) -> (a' (H, (updateR R rd (lookupR R rs)))))
           -> (Infer ct a' b)
           -> (Infer ct a (seq (mov rd rs) b))
 | I_bgt : (ct:codeheapty; a,a',a1:assert; rs,rt:reg; f:lab; b:iseq)
           (lookupCT ct f a1)
           -> ((H:heap; R:rfile)
               (le (lookupR R rs) (lookupR R rt))
               -> (a (H, R)) -> (a' (H, R)))
           -> ((H:heap; R:rfile)
               (gt (lookupR R rs) (lookupR R rt))
               -> (a (H, R)) -> (a1 (H, R)))
           -> (Infer ct a' b)
           -> (Infer ct a (seq (bgt rs rt f) b))
 | I_jd  : (ct:codeheapty; a,a1:assert; f:lab)
           (lookupCT ct f a1)
           -> ((s:state) (a s) -> (a1 s))
           -> (Infer ct a (jd f))
 | I_jmp : (ct:codeheapty; a:assert; r:reg)
           ((H:heap; R:rfile)
            (a (H, R)) -> (EXT a1:assert |
              (lookupCT ct (lookupR R r) a1) /\ (a1 (H, R))))
           -> (Infer ct a (jmp r))
 | ...

(* well-formed code heap *)
Inductive WF_CH_0 : codeheapty -> codeheapty -> codeheap -> Prop :=
 | wf_c_emp  : (ct0:codeheapty) (WF_CH_0 ct0 emptyCT emptyC)
 | wf_c_cons : (ct0,ct:codeheapty;ch:codeheap;f:lab;a:assert;b:iseq)
               (Infer ct0 a b)
               -> (WF_CH_0 ct0 ct ch)
               -> (WF_CH_0 ct0 (consCT f a ct) (consC f b ch)).

Inductive WF_CH : codeheapty -> codeheap -> Prop :=
 | wf_c : (ct:codeheapty; ch:codeheap)
          (WF_CH_0 ct ct ch) -> (WF_CH ct ch).

(* well-formed program *)
Inductive InferP : codeheapty -> assert -> prog -> Prop :=
  I_prog : (ct:codeheapty; a:assert; ch:codeheap; s:state; b:iseq)
           (WF_CH ct ch) -> (Infer ct a b) -> (a s)
           -> (InferP ct a (ch, (s, b))).
```

Fig. 10. Coq encoding of CAP inference rules.

```
Lemma Progress : (ct:codeheapty; a:assert)
 (ch:codeheap; s:state; I:iseq)
 (InferP ct a (ch, (s, I))) -> (EX P:prog | (Step (ch, (s, I)) P)).
```

```
Lemma Preservation : (ct:codeheapty; a:assert)
 (ch:codeheap; s:state; I:iseq; P:prog)
 (InferP ct a (ch, (s, I))) -> (Step (ch, (s, I)) P)
 -> (EXT a':assert | (InferP ct a' P)).
```

### 6.4. *Proof construction for libraries and programs*

The code, specification templates and proof outlines of the library routines free and malloc and the example program copy can be found in Appendix A. Once the behavior of common operations is factored out as lemmas, proof construction becomes largely routine work. During our implementation, it took only two weeks for a single graduate student, who was a beginner at using the Coq proof assistant and hence only used the most basic Coq tactics, to construct all the proofs for free and malloc, including proving all the related lemmas.

Now we conclude this section by giving the declarations of some lemmas that are commonly used in proving the well-formedness of the free and malloc routines. In these declarations, `star` is the encoding of the separating conjunction operator $*$ (i.e., `(star a1 a2)` stands for $a_1 * a_2$), `MBlk` and `MBlkLst` are the encodings of the corresponding assertions defined in Section 4, and `(hwrite H p q')` returns a heap which is the original heap `H` updated so that the location `p` stores the value `q'`.

```
(* symmetricity of separating conjunction *)
Lemma star_sym : (a1,a2:assert; s:state)
 (star a1 a2 s) -> (star a2 a1 s).
```

```
(* to update the ``link" field of a memory block header *)
Lemma Write_MBlk_lnk : (H:heap; R:rfile; p,q,q':addr; siz:nat)
 (MBlk (plus (2) p) q siz (H,R))
  -> (MBlk (plus (2) p) q' siz ((hwrite H p q'),R)).
```

```
(* to compose a longer MBlkLst by adding an MBlk at the end *)
Lemma Compose_MBlkLst_MBlk_to_MBlkLst :
 (n:nat; s:state; flist,p,q:addr; siz:nat)
 (star (MBlkLst n flist p) (MBlk (plus (2) p) q siz) s)
  -> ((lt p q) \/ (q=pnil))     (* pnil -- the null pointer *)
  -> (MBlkLst (S n) flist q s).
```

## 7. Related work and future work

*Dynamic storage allocation.* Wilson et al. [22] categorized allocators based on *strategies* (which attempt to exploit regularities in program behavior), *placement policies* (which decide where to allocate and return blocks in memory), and *mechanisms* (which involve the algorithms and data structures that implement the policy). We believe that the most tricky part in certifying various allocators is on the low-level mechanisms, rather than the high-level strategies and policies. Most allocators share some subsidiary techniques, such as *splitting* and *coalescing*. Although we only provide a single allocation library implementing a particular policy, the general idea used to certify the techniques of splitting and coalescing can be applied to implement other policies.

*Hoare logic.* Our logical reasoning about memory properties directly follows Reynolds' separation logic [19,18]. However, being at an assembly level, CAP has some advantages in the context of mechanical proof-checking. CAP provides a fixed number of registers so the dynamic state is easier to model than using infinite number of variables, and programs are free of variable shadowing. Being at a lower-level implies that the compiler to the final machine code is easier to build, hence it involves a smaller trusted computing base (TCB). Embedding our language for assertions directly in CiC is also crucial for mechanical proof-checking and PCC. Another difference is that we establish the soundness property of our language using a syntactic approach.

Filliâtre [6,5] developed a software certification tool *Why* which takes annotated programs as input and outputs proof obligations based on Hoare logic for proof assistants Coq and PVS. It is possible to apply *Why* in the PCC framework, because the proof obligation generator is closely related to the verification condition generator of PCC. However, it is less clear how to apply *Why* to Foundational PCC because the proof obligation generator would have to be trusted. On the other hand, if *Why* is applied to certify memory management, it is very likely to hit problems such as expressing inductively defined assertions. Our treatment of assertions in mechanical proof-checking can be used to help.

*Certifying compilation.* This paper is largely complementary to existing work on certifying compilation [16,12,3,1]. Existing work has only focused on programs whose safety proofs can be automatically generated. In contrast, we support general properties and partial program correctness, but we rely on the programmer to construct the proof. Nevertheless, we believe this is necessary for reasoning about program correctness. Automatic proof construction is infeasible because the problem in general is undecidable. Our language can be used to formally present the reasonings of a programmer. With the help of proof assistants, proof construction is not difficult, and the result can be mechanically checked.

*Future work.* CAP is a small, generic, yet fairly realistic language. It suggests an elegant framework for code certification. However, admittedly, the simplicity of CAP limits its expressiveness in the context of modularity. To get around this limitation, we designed the library specs and their well-formedness theorems to be templates which can be instantiated according to the user programs. The modularity we achieved here is sub-optimal in the sense that, ideally, a library routine should be given a unique and full-fledged spec and well-formedness theorem. Supporting modularity in this optimal sense is hard, especially with higher-order code pointers involved. We are currently working on the next generation of CAP which addresses these issues.

Additionally, our further development of CAP will involve its application to a real machine model, for example, the Intel x86 architecture, instead of the idealistic machine used in this paper. This will allow us to produce "runnable" packages of PCC.

```
free : $\{(\mathsf{MBlk}\ fptr\ \_\ \_) * (\mathsf{Good}\ flist)$
         $*Pred \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}\}$
       subi $hp, fptr, 2$;
       movi $prev$, nil;
       mov $p, flist$;
       $\{(\mathsf{MBlk}\ (hp+2)\ \_\ \_) * (\mathsf{MidL}\ flist\ prev\ p)$
         $*Pred \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (prev = \mathtt{nil})\}$
       jd iter;

next : $\{(\mathsf{MBlk}\ (hp+2)\ \_\ \_) * (\mathsf{MidL}\ flist\ prev\ p)$
         $*Pred \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (p \neq \mathtt{nil})$
         $\wedge(prev < hp \vee prev = \mathtt{nil})\}$
       bgt $p, hp$, tryh;
       $\{(\mathsf{MBlk}\ (hp+2)\ \_\ \_) * (\mathsf{MidL}\ flist\ prev\ p)$
         $*Pred \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (p < hp) \wedge (p \neq \mathtt{nil})\}$
       mov $prev, p$;
       ld $p, p(\mathtt{link})$;
       $\{(\mathsf{MBlk}\ (hp+2)\ \_\ \_) * (\mathsf{MidL}\ flist\ prev\ p)$
         $*Pred \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (prev < hp)\}$
       jd iter;

njhi : $\{(\mathsf{MBlk}\ (hp+2)\ \_\ s) * (\mathsf{MidL}\ flist\ prev\ p)$
         $*Pred \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (p > hp + s)$
         $\wedge(prev < hp \vee prev = \mathtt{nil})\}$
       st $hp(\mathtt{link}), p$;
       $\{(\mathsf{MBlk}\ (hp+2)\ p\ s) * (\mathsf{EndL}\ flist\ prev\ p)$
         $*(\mathsf{MBlkLst}\ n\ p\ \mathtt{nil}) * Pred$
         $\wedge\mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (hp < p)$
         $\wedge(prev < hp \vee prev = nil)\}$
       jd tryl;

tryl : $\{(\mathsf{EndL}\ flist\ prev\ \_) * Pred$
         $*(\mathsf{MBlkLst}\ (n+1)\ hp\ \mathtt{nil})$
         $\wedge\mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (prev < hp \vee prev = \mathtt{nil})\}$
       bgti $prev$, nil, lnkl;
       $\{(\mathsf{EndL}\ flist\ prev\ \_) * Pred$
         $*(\mathsf{MBlkLst}\ (n+1)\ hp\ \mathtt{nil})$
         $\wedge\mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (prev = \mathtt{nil})\}$
       mov $flist, hp$;
       $\{(\mathsf{Good}\ flist) * Pred \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}\}$
       jmp $\mathtt{r}_0$;

njlo : $\{(\mathsf{MBlkLst}\ m\ flist\ prev)$
         $*(\mathsf{MBlk}\ (prev+2)\ \_\ prevsize)$
         $*(\mathsf{MBlkLst}\ (n+1)\ hp\ \mathtt{nil}) * Pred$
         $\wedge\mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (prev < hp)$
         $\wedge(prev + prevsize < hp)\}$
       st $prev(\mathtt{link}), hp$;
       $\{(\mathsf{Good}\ flist) * Pred \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}\}$
       jmp $\mathtt{r}_0$;

iter : $\{(\mathsf{MBlk}\ (hp+2)\ \_\ \_) * Pred$
         $*(\mathsf{MidL}\ flist\ prev\ p) \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}$
         $\wedge(prev < hp \vee prev = \mathtt{nil})\}$
       bgti $p$, nil, next;
       st $hp(\mathtt{link}), p$;
       jd tryl;

tryh : $\{(\mathsf{MBlk}\ (hp+2)\ \_\ \_) * Pred$
         $*(\mathsf{MidL}\ flist\ prev\ p) \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}$
         $\wedge(p \neq \mathtt{nil}) \wedge (hp < p)$
         $\wedge(prev < hp \vee prev = nil)\}$
       ld $cursize, hp(\mathtt{size})$;
       add $curend, hp, cursize$;
       bgt $p, curend$, njhi;
       $\{(\mathsf{MBlk}\ (hp+2)\ \_\ cursize) * Pred$
         $*(\mathsf{MidL}\ flist\ prev\ p) \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}$
         $\wedge(prev < hp \vee prev = \mathtt{nil})$
         $\wedge(p = hp + cursize \neq \mathtt{nil})\}$
       ld $psize, p(\mathtt{size})$;
       add $newsize, cursize, psize$;
       st $hp(\mathtt{size}), newsize$;
       ld $plink, p(\mathtt{link})$;
       st $hp(\mathtt{link}), plink$;
       $\{(\mathsf{MBlk}\ (hp+2)\ q\ \_) * Pred$
         $*(\mathsf{EndL}\ flist\ prev\ p)$
         $*(\mathsf{MBlkLst}\ n\ q\ \mathtt{nil}) \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}$
         $\wedge(prev < hp \vee prev = nil)$
         $\wedge(hp < q \vee q = \mathtt{nil})\}$
       jd tryl;

lnkl : $\{(\mathsf{MBlkLst}\ m\ flist\ prev)$
         $*(\mathsf{MBlk}\ (prev+2)\ \_\ s) \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}$
         $*(\mathsf{MBlkLst}\ (n+1)\ hp\ \mathtt{nil}) * Pred$
         $\wedge(prev < hp) \wedge (prev \neq \mathtt{nil})\}$
       ld $prevsize, prev(\mathtt{size})$;
       add $prevend, prev, prevsize$;
       bgt $hp, prevend$, njlo;
       $\{(\mathsf{MBlkLst}\ m\ flist\ prev)$
         $*(\mathsf{MBlk}\ (prev+2)\ \_\ prevsize)$
         $*(\mathsf{MBlkLst}\ (n+1)\ hp\ \mathtt{nil}) * Pred$
         $\wedge\mathbb{R}(\mathtt{r}_0) = \mathtt{f} \wedge (prev < hp)$
         $\wedge(prev + prevsize = hp)\}$
       ld $cursize, hp(\mathtt{size})$;
       add $newsize, prevsize, cursize$;
       ld $curlink, hp(\mathtt{link})$;
       st $prev(\mathtt{size}), newsize$;
       st $prev(\mathtt{link}), curlink$;
       $\{(\mathsf{Good}\ flist) * Pred \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{f}\}$
       jmp $\mathtt{r}_0$;
```

where link $\equiv 0$, size $\equiv 1$ and nil $\equiv 0$; variables are shorthands for registers.

Fig. 11. Annotated program of free.

```
malloc : $\{Pred * (\mathsf{Good}\ flist)$
      $\wedge(nsize = s_0 > 0) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
     addi $bsize, nsize, 2$;
     jd init;

init : $\{Pred * (\mathsf{Good}\ flist)$
     $\wedge(0 < s_0 + 2 \le bsize) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
    movi $prev$, nil;
    mov $p, flist$;
    $\{Pred * (\mathsf{MidL}\ flist\ prev\ p)$
     $\wedge(0 < s_0 + 2 \le bsize) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
    jd miter;

miter : $\{Pred * (\mathsf{MidL}\ flist\ prev\ p)$
      $\wedge(0 < s_0 + 2 \le bsize) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
     bgti $p$, nil, comp;
     bgt $NALLOC, bsize$, mod;
     $\{Pred * (\mathsf{Good}\ flist)$
      $\wedge(0 < s_0 + 2 \le bsize) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
     mov $bbsize, bsize$;
     jd more;

comp : $\{Pred * (\mathsf{MidL}\ flist\ prev\ p) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}$
    $\wedge(0 < s_0 + 2 \le bsize) \wedge (p \ne \mathtt{nil})\}$
    ld $psize, p(\mathtt{size})$;
    addi $ebsize, bsize, 2$;
    bgt $psize, ebsize$, split;
    $\{Pred * (\mathsf{EndL}\ flist\ prev\ p)$
     $*(\mathsf{MBlk}\ (p+2)\ q\ psize) * (\mathsf{MBlkLst}\ n\ q\ \mathtt{nil})$
     $\wedge(0 < s_0 + 2 \le bsize) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}$
     $\wedge(p < q \vee q = \mathtt{nil})\}$
    bgt $bsize, psize$, mnext;
    $\{Pred * (\mathsf{EndL}\ flist\ prev\ p)$
     $*(\mathsf{MBlk}\ (p+2)\ q\ psize)$
     $*(\mathsf{MBlkLst}\ n\ q\ \mathtt{nil})$
     $\wedge(s_0 + 2 \le bsize \le psize) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}$
     $\wedge(p < q \vee q = \mathtt{nil})\}$
    ld $plink, p(\mathtt{link})$;
    bgti $prev$, nil, lprv;
    $\{Pred * (\mathsf{EndL}\ flist\ \mathtt{nil}\ p)$
     $*(\mathsf{MBlk}\ (p+2)\ plink\ psize)$
     $*(\mathsf{MBlkLst}\ n\ plink\ \mathtt{nil})$
     $\wedge(s_0 + 2 \le psize) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
    mov $flist, plink$;
    $\{Pred * (\mathsf{MBlk}\ (p+2)\ plink\ psize)$
     $*(\mathsf{MBlkLst}\ n\ flist\ \mathtt{nil})$
     $\wedge(s_0 + 2 \le psize) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
    jd retptr;

retptr : $\{Pred * (\mathsf{Good}\ flist) * (\mathsf{MBlk}\ (p+2)\ \_\ s)$
      $\wedge(s_0 + 2 \le s) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
     addi $mptr, p, 2$;
     $\{Pred' * (\mathsf{Good}\ flist) * (\mathsf{MBlk}\ mptr\ \_\ s)$
      $\wedge(s_0 + 2 \le s) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
     jmp $\mathtt{r}_1$;

mod : $\{Pred * (\mathsf{Good}\ flist) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\wedge$
    $(0 < s_0 + 2 \le bsize < NALLOC)\}$
   mov $bbsize, NALLOC$;
   jd more;

split : $\{Pred * (\mathsf{EndL}\ flist\ prev\ p)$
     $*(\mathsf{MBlk}\ (p+2)\ q\ psize)$
     $*(\mathsf{MBlkLst}\ n\ q\ \mathtt{nil}) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}$
     $\wedge(0 < s_0 + 2 \le bsize < psize - 2)$
     $\wedge(p < q \vee q = \mathtt{nil})\}$
    sub $psize, psize, bsize$;
    st $p(\mathtt{size}), psize$;
    add $p, p, psize$;
    st $p(\mathtt{size}), bsize$;
    $\{Pred * (\mathsf{EndL}\ flist\ prev\ p')$
     $*(\mathsf{MBlk}\ (p'+2)\ q\ (psize - bsize))$
     $*(\mathsf{MBlk}\ (p+2)\ \_\ bsize)$
     $*(\mathsf{MBlkLst}\ n\ q\ \mathtt{nil}) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}$
     $\wedge(0 < s_0 + 2 \le bsize < psize - 2)$
     $\wedge(p' < q \vee q = \mathtt{nil})\}$
    jd retptr;

mnext : $\{Pred * (\mathsf{EndL}\ flist\ prev\ p)$
     $*(\mathsf{MBlk}\ (p+2)\ q\ s)$
     $*(\mathsf{MBlkLst}\ n\ q\ \mathtt{nil}) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}$
     $\wedge(0 < s_0 + 2 \le bsize)$
     $\wedge(p < q \vee q = \mathtt{nil})\}$
    mov $prev, p$;
    ld $p, p(\mathtt{link})$;
    $\{Pred * (\mathsf{EndL}\ flist\ prev\ p)$
     $*(\mathsf{MBlkLst}\ n\ p\ \mathtt{nil}) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}$
     $\wedge(0 < s_0 + 2 \le bsize)\}$
    jd miter;

lprv : $\{Pred * (\mathsf{EndL}\ flist\ prev\ p)$
     $*(\mathsf{MBlk}\ (p+2)\ plink\ s)*$
     $(\mathsf{MBlkLst}\ n\ plink\ \mathtt{nil}) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}$
     $\wedge(s_0 + 2 \le s) \wedge (prev \ne \mathtt{nil})$
     $\wedge(p < plink \vee plink = \mathtt{nil})\}$
    st $prev(\mathtt{link}), plink$;
    $\{Pred * (\mathsf{EndL}\ flist\ prev\ plink)$
     $*(\mathsf{MBlk}\ (p+2)\ plink\ s)$
     $*(\mathsf{MBlkLst}\ n\ plink\ \mathtt{nil})$
     $\wedge(s_0 + 2 \le s) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}\}$
    jd retptr;

more : $\{Pred * (\mathsf{Good}\ flist) \wedge \mathbb{R}(\mathtt{r}_1) = \mathtt{f}$
    $\wedge(0 < s_0 + 2 \le bsize \le bbsize)\}$
   alloc $newp[bbsize]$;
   st $newp(\mathtt{size}), bbsize$;
   addi $fptr, newp, 2$;
   movi $\mathtt{r}_0$, init;
   $\{(\mathsf{Good}\ flist) * (\mathsf{MBlk}\ fptr\ \_\ bbsize)$
    $*Pred \wedge (0 < s_0 + 2 \le bsize)$
    $\wedge\mathbb{R}(\mathtt{r}_1) = \mathtt{f} \wedge \mathbb{R}(\mathtt{r}_0) = \mathtt{init}\}$
   jd free;
```

where $\mathtt{link} \equiv 0$, $\mathtt{size} \equiv 1$ and $\mathtt{nil} \equiv 0$; variables are shorthands for registers.

Fig. 12. Annotated program of malloc.

```
copy : $\{\exists\alpha.\,(\mathsf{Slist}\ \alpha\ src\ \mathtt{nil}) * (\mathsf{Good}\ flist)$
         $\wedge(\alpha = \alpha_0)\}$
       movi $tgt, \mathtt{nil}$;
       movi $cprev, \mathtt{nil}$;
       jd test;

test : $\{(((cprev = \mathtt{nil})$
          $\supset \exists\alpha.\,(\mathsf{Slist}\ \alpha\ src\ \mathtt{nil}) \wedge (\alpha = \alpha_0)$
            $\wedge(tgt = \mathtt{nil}))$
         $\wedge((cprev \neq \mathtt{nil})$
           $\supset \exists\alpha.\,\exists\beta.\,\exists b.\,(\beta\cdot b\cdot\alpha = \alpha_0)$
             $\wedge(\mathsf{Slist}\ \alpha\ src\ \mathtt{nil}) * (\mathsf{Slist}\ \beta\ tgt\ cprev)$
             $*(\mathsf{Pair}\ cprev\ b\ src)))$
         $*(\mathsf{Good}\ flist)\}$
       bgti $src, \mathtt{nil}, \mathtt{nxt0}$;
       jd halt;

nxt0 : $\{(((cprev = \mathtt{nil})$
          $\supset \exists a.\,\exists\alpha.\,\exists i.\,(\mathsf{Pair}\ src\ a\ i) * (\mathsf{Slist}\ \alpha\ i\ \mathtt{nil})$
            $\wedge(a\cdot\alpha = \alpha_0) \wedge (tgt = \mathtt{nil}))$
         $\wedge((cprev \neq \mathtt{nil})$
           $\supset \exists a.\,\exists\alpha.\,\exists\beta.\,\exists b.\,\exists i.\,(\beta\cdot b\cdot a\cdot\alpha = \alpha_0)$
             $\wedge(\mathsf{Pair}\ src\ a\ i) * (\mathsf{Slist}\ \alpha\ i\ \mathtt{nil})$
             $*(\mathsf{Slist}\ \beta\ tgt\ cprev)$
             $*(\mathsf{Pair}\ cprev\ b\ src)))$
         $*(\mathsf{Good}\ flist)\}$
       movi $nsize, 2$;
       movi $\mathtt{r}_1, \mathtt{nxt1}$;
       $\{(((cprev = \mathtt{nil})$
          $\supset \exists a.\,\exists\alpha.\,\exists i.\,(\mathsf{Pair}\ src\ a\ i) * (\mathsf{Slist}\ \alpha\ i\ \mathtt{nil})$
            $\wedge(a\cdot\alpha = \alpha_0) \wedge (tgt = \mathtt{nil}))$
         $\wedge((cprev \neq \mathtt{nil})$
           $\supset \exists a.\,\exists\alpha.\,\exists\beta.\,\exists b.\,\exists i.\,(\beta\cdot b\cdot a\cdot\alpha = \alpha_0)$
             $\wedge(\mathsf{Pair}\ src\ a\ i) * (\mathsf{Slist}\ \alpha\ i\ \mathtt{nil})$
             $*(\mathsf{Slist}\ \beta\ tgt\ cprev)$
             $*(\mathsf{Pair}\ cprev\ b\ src)))$
         $*(\mathsf{Good}\ flist) \wedge (nsize = 2) \wedge (\mathbb{R}(\mathtt{r}_1) = \mathtt{nxt1})\}$
       jd malloc;

lnkp : $\{(((cprev \neq \mathtt{nil})$
          $\wedge \exists a.\,\exists\alpha.\,\exists\beta.\,\exists b.\,(\mathsf{Pair}\ mptr\ a\ src)$
            $*(\mathsf{Slist}\ \alpha\ src\ \mathtt{nil}) * (\mathsf{Slist}\ \beta\ tgt\ cprev)$
            $*(\mathsf{Pair}\ cprev\ b\ fptr) \wedge (\beta\cdot b\cdot a\cdot\alpha = \alpha_0)))$
         $*(\mathsf{Good}\ flist)\}$
       st $cprev(1), mptr$;
       mov $cprev, mptr$;
       $\{(((cprev \neq \mathtt{nil})$
          $\wedge \exists a.\,\exists\alpha.\,\exists\beta.\,(\mathsf{Pair}\ cprev\ a\ src)$
            $*(\mathsf{Slist}\ \alpha\ src\ \mathtt{nil})$
            $*(\mathsf{Slist}\ \beta\ tgt\ cprev) \wedge (\beta\cdot a\cdot\alpha = \alpha_0)))$
         $*(\mathsf{Good}\ flist)\}$
       jd test;

halt : $\{\exists\beta.\,(\mathsf{Slist}\ \beta\ tgt\ \mathtt{nil}) * (\mathsf{Good}\ flist)$
         $\wedge(\beta = \alpha_0)\}$
       jd halt;
where nil ≡ 0; variables are shorthands for registers.

nxt1 : $\{(((cprev = \mathtt{nil})$
          $\supset \exists a.\,\exists\alpha.\,\exists i.\,(\mathsf{Pair}\ src\ a\ i)$
            $*(\mathsf{Slist}\ \alpha\ i\ \mathtt{nil}) \wedge (a\cdot\alpha = \alpha_0)$
            $\wedge(tgt = \mathtt{nil}))$
         $\wedge((cprev \neq \mathtt{nil})$
           $\supset \exists(a,\alpha,\beta,b,i).\,(\beta\cdot b\cdot a\cdot\alpha = \alpha_0)$
             $\wedge(\mathsf{Pair}\ src\ a\ i) * (\mathsf{Slist}\ \alpha\ i\ \mathtt{nil})$
             $*(\mathsf{Slist}\ \beta\ tgt\ cprev)$
             $*(\mathsf{Pair}\ cprev\ b\ src)))$
         $*(\mathsf{Good}\ flist) * (\mathsf{MBlk}\ mptr\ \_\ siz)$
         $\wedge(siz \geq 4)\}$
       ld $temp, src(0)$;
       st $mptr(0), temp$;
       mov $fptr, src$;
       ld $src, src(1)$;
       st $mptr(1), src$;
       movi $\mathtt{r}_0, \mathtt{nxt2}$
       $\{(((cprev = \mathtt{nil})$
          $\supset \exists a.\,\exists\alpha.\,(\mathsf{Pair}\ mptr\ a\ src)$
            $*(\mathsf{Slist}\ \alpha\ src\ \mathtt{nil}) \wedge (a\cdot\alpha = \alpha_0)$
            $\wedge(tgt = \mathtt{nil}))$
         $\wedge((cprev \neq \mathtt{nil})$
           $\supset \exists a.\,\exists\alpha.\,\exists\beta.\,\exists b.\,(\beta\cdot b\cdot a\cdot\alpha = \alpha_0)$
             $\wedge(\mathsf{Pair}\ mptr\ a\ src)$
             $*(\mathsf{Slist}\ \alpha\ src\ \mathtt{nil})$
             $*(\mathsf{Slist}\ \beta\ tgt\ cprev)$
             $*(\mathsf{Pair}\ cprev\ b\ fptr)))$
         $*(\mathsf{MBlk}\ fptr\ \_\ \_) * (\mathsf{Good}\ flist)$
         $\wedge(\mathbb{R}(\mathtt{r}_0) = \mathtt{nxt2})\}$
       jd free;

nxt2 : $\{(((cprev = \mathtt{nil})$
          $\supset \exists a.\,\exists\alpha.\,(\mathsf{Pair}\ mptr\ a\ src)$
            $*(\mathsf{Slist}\ \alpha\ src\ \mathtt{nil}) \wedge (a\cdot\alpha = \alpha_0)$
            $\wedge(tgt = \mathtt{nil}))$
         $\wedge((cprev \neq \mathtt{nil})$
           $\supset \exists a.\,\exists\alpha.\,\exists\beta.\,\exists b.\,(\beta\cdot b\cdot a\cdot\alpha = \alpha_0)$
             $\wedge(\mathsf{Pair}\ mptr\ a\ src)$
             $*(\mathsf{Slist}\ \alpha\ src\ \mathtt{nil})$
             $*(\mathsf{Slist}\ \beta\ tgt\ cprev)$
             $*(\mathsf{Pair}\ cprev\ b\ fptr)))$
         $*(\mathsf{Good}\ flist)\}$
       bgti $cprev, \mathtt{nil}, \mathtt{lnkp}$;
       $\{\exists a.\,\exists\alpha.\,(\mathsf{Pair}\ mptr\ a\ src)$
         $*(\mathsf{Slist}\ \alpha\ src\ \mathtt{nil}) \wedge (a\cdot\alpha = \alpha_0)$
         $\wedge(tgt = \mathtt{nil})$
         $*(\mathsf{Good}\ flist) \wedge (cprev = \mathtt{nil})\}$
       mov $tgt, mptr$;
       mov $cprev, tgt$;
       $\{\exists a.\,\exists\alpha.\,(\mathsf{Pair}\ cprev\ a\ src)$
         $*(\mathsf{Slist}\ \alpha\ src\ \mathtt{nil}) \wedge (a\cdot\alpha = \alpha_0)$
         $\wedge(cprev = tgt)$
         $*(\mathsf{Good}\ flist) \wedge (cprev \neq \mathtt{nil})\}$
       jd test;
```

Fig. 13. Annotated program of copy: copies a null-terminated list from *src* to *tgt*.

By exploring the design space that lies between Hoare-logic and type systems, we intend to begin modeling types as assertion macros in CAP to ease the tasks associated with certifying code. For instance, a useful macro would be the type of a memory block (MBlk). With lemmas ("typing rules") on how this macro interacts with commands, users can propagate it conveniently. If one is only interested in simple properties (e.g., operations are performed only on allocated blocks), it may be possible to achieve proof construction with little or no user interaction.

In the future, it would be interesting to develop high-level (e.g., C-like or Java-like) surface languages with similar explicit specifications so that programs are written at a higher-level. "Proof-preserving" compilation from those languages to CAP may help retain a small trusted computing base.

## 8. Conclusion

Existing certifying compilers have only focused on programs whose safety proofs can be automatically generated. Adopting a complementary approach, we explore in this paper Complementary to these works, we explored in this paper how to certify general properties and program correctness in the PCC framework, letting programmers develop proofs semi-automatically with help of a proof assistant. In particular, we have presented a certified library for dynamic storage allocation—an area hard to handle using conventional type systems. The logical reasoning about memory management largely follows the style of separation logic. In general, it seems that working towards an interface between Hoare-logic reasoning and type systems will yield interesting results in the context of PCC.

## Appendix A. Annotated programs

The programs in this section have interspersed assertions in the code blocks, which does not strictly follow the syntax of CAP presented in the main portion of the paper. These annotations really correspond to the assertions $a'$ that must be provided in the CAP inference rules for well-formed instruction sequences. The assertion at the very beginning of each code block label is what that label would be mapped to in the code heap spec (see Figs. 11–13).

## References

[1] A.W. Appel, Foundational proof-carrying code, in: Proc. 16th Annu. IEEE Symp. on Logic in Computer Science, June 2001, pp. 247–258.

[2] A.W. Appel, E.W. Felten, Models for security policies in proof-carrying code, Technical Report CS-TR-636-01, Department of Computer Science, Princeton University, March 2001.

[3] C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, K. Cline, A certifying compiler for Java, in: Proc. 2000 ACM Conf on Programming Language Design and Implementation, ACM Press, New York, 2000, pp. 95–107.

[4] T. Coquand, G. Huet, The calculus of constructions, Inform. and Comput. 76 (1988) 95–120.

[5] J.-C. Filliâtre, The WHY certification tool, tutorial and reference manual, http://why.lri.fr/, July 2002.
[6] J.-C. Filliâtre, Verification of non-functional programs using interpretations in type theory, J. Funct. Programming 13 (4) (2003) 709–745.
[7] C.A.R. Hoare, An axiomatic basis for computer programming, Comm. ACM 12 (10) (1969) 576–580.
[8] C.A.R. Hoare, Proof of a program: FIND, Comm. ACM 14 (1) (1971) 39–45.
[9] W.A. Howard, The formulae-as-types notion of constructions, in: J.P. Seldin, J.R. Hindley (Eds.), To H.B.Curry: Essays on Computational Logic, Lambda Calculus and Formalism, Academic Press, New York, 1980.
[10] B.W. Kernighan, D.M. Ritchie, The C Programming Language, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1988.
[11] D.E. Kunth, The Art of Computer Programming, 2nd ed., vol. 1, Addison-Wesley, Reading, MA, 1973.
[12] G. Morrisett, D. Walker, K. Crary, N. Glew, From system F to typed assembly language, in: Proc. 25th ACM Symp. on Principles of Programming Languages, ACM Press, New York, January 1998, pp. 85–97.
[13] G. Necula, Proof-carrying code, in: Proc. 24th ACM Symp. on Principles of Programming Languages, ACM Press, New York, January 1997, pp. 106–119.
[14] G. Necula, Compiling with proofs, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, September 1998.
[15] G. Necula, P. Lee, Safe kernel extensions without run-time checking, in: Proc. Second USENIX Symp. on Operating System Design and Implementation, 1996, pp. 229–243.
[16] G. Necula, P. Lee, The design and implementation of a certifying compiler, in: Proc. 1998 ACM Conf. on Programming Languages Design and Implementation, New York, 1998, pp. 333–344.
[17] C. Paulin-Mohring, Inductive definitions in the system Coq—rules and properties, in: M. Bezen, J. Groote (Eds.), Proc. TLCA, Lecture Notes in Computer Science, vol. 664, Springer, Berlin, 1993.
[18] J.C. Reynolds, Lectures on reasoning about shared mutable date structure, IFIP Working Group 2.3 School/Seminar on State-of-the Art Program Design, Using Logic, Tandil, Argentina, September 6–13, 2000.
[19] J.C. Reynolds, Separation logic: a logic for shared mutable data structures, in: Proc. Seventeenth Annu. IEEE Symp. on Logic in Computer Science, Los Alamitos, California, IEEE Computer Society, Silver Spring, MD, 2002.
[20] The Coq Development Team, The Coq proof assistant reference manual, The Coq release v7.1, October 2001.
[21] The FLINT Project, Coq implementation for certified dynamic storage allocation, http://flint.cs.yale.edu/flint/publications/cdsa.html, October 2002.
[22] P.R. Wilson, M.S. Johnstone, M. Neely, D. Boles, Dynamic storage allocation: a survey and critical review, in: Proc. Internat. Workshop on Memory Management, Kinross Scotland, UK, 1995.
[23] A.K. Wright, M. Felleisen, A syntactic approach to type soundness, Inform. and Comput. 115 (1) (1994) 38–94.