Applying Abstract Stobjs to Processor Modeling

Shilpi Goel
Warren Hunt
Matt Kaufmann

ACL2 Seminar, October 2, 2012
Outline

Introduction

Processor Modeling

Abstract Stobjs

Two Advantages of Abstract Stobjs
  Eliminating Hypotheses
  Avoiding Expensive Guard-Checking

Proof by Symbolic Execution (GL)
  GL: Introduction
  Proof of Correctness of the Y86 Popcount Program

Conclusion
Outline

Introduction

Processor Modeling

Abstract Stobjs

Two Advantages of Abstract Stobjs
  Eliminating Hypotheses
  Avoiding Expensive Guard-Checking

Proof by Symbolic Execution (GL)
  GL: Introduction
  Proof of Correctness of the Y86 Popcount Program

Conclusion
Introduction

**Goal:** Illustrate abstract stobjs and their application to processor modeling
Introduction: Processor Models

ACL2 processor models:

- (Hunt) Bryant’s Y86
- (Hunt, Kaufmann) Early X86 model with space-efficient memory model
- (Hunt, Goel) New X86 model
- (Kaufmann) Abstract Stobjs: Early X86 model
- (Goel) Y86 with space-efficient memory model
- (Goel) [In progress] Abstract stobjs: Y86 (towards X86)
- (Krug) [In progress] Paging in X86
(Moore, others) Using rewriting

(Swords) Symbolic Execution using GL

- Hunt and Kaufmann used GL for code proofs on a non-stobj Y86 model.
- How can we use GL to do proofs about large stobj memories efficiently?

Abstract Stobjs!
Introduction: Code Proofs

- (Moore, others) Using rewriting
- (Swords) Symbolic Execution using GL
  - Hunt and Kaufmann used GL for code proofs on a non-stobj Y86 model.
  - How can we use GL to do proofs about large stobj memories efficiently?

Abstract Stobjs!
Introduction: Code Proofs

- (Moore, others) Using rewriting
- (Swords) Symbolic Execution using GL
- Hunt and Kaufmann used GL for code proofs on a non-stobj Y86 model.
- How can we use GL to do proofs about large stobj memories efficiently?

Abstract Stobjs!
Introduction: Code Proofs

- (Moore, others) Using rewriting
- (Swords) Symbolic Execution using GL
  - Hunt and Kaufmann used GL for code proofs on a non-stobj Y86 model.
  - How can we use GL to do proofs about large stobj memories efficiently?
Introduction: Code Proofs

- (Moore, others) Using rewriting
- (Swords) Symbolic Execution using GL
  - Hunt and Kaufmann used GL for code proofs on a non-stobj Y86 model.
  - How can we use GL to do proofs about large stobj memories efficiently?

Abstract Stobjs!
Outline

Introduction

Processor Modeling

Abstract Stobjs

Two Advantages of Abstract Stobjs
   Eliminating Hypotheses
   Avoiding Expensive Guard-Checking

Proof by Symbolic Execution (GL)
   GL: Introduction
   Proof of Correctness of the Y86 Popcount Program

Conclusion
Before we model the ISA of the Y86, we need to define its state.

The Y86 processor state (which we call $y86$c) is defined using a single threaded object, or stobj.

Stobjs in ACL2 are mutable objects that have applicative semantics.
Processor Modeling: Y86 State

Before we model the ISA of the Y86, we need to define its state.

The Y86 processor state (which we call $y86$c) is defined using a single threaded object, or stobj.

Stobjs in ACL2 are mutable objects that have applicative semantics.
Before we model the ISA of the Y86, we need to define its state.

The Y86 processor state (which we call $y86$c) is defined using a *single threaded object*, or *stobj*.

*Stobjs* in ACL2 are mutable objects that have applicative semantics.
Processor Modeling: Processor State

(defstobj y86$c

;; The program counter.
(eip$c :type (unsigned-byte 32)
   :initially 0)
...

;; The memory model: space-efficient implementation
(mem-table :type (array (unsigned-byte 32)
                           (* mem-table-size*))
           :initially 1
           :resizable nil)

(mem-array :type (array (unsigned-byte 8)
                         (* initial-mem-array-length*))
            :initially 0
            :resizable t)

(mem-array-next-addr :type (integer 0 4294967296)
                      :initially 0)
...
:renaming ((y86$cp y86$cp-pre))
)
We have renamed the recognizer for the $y86$c stobj from $y86$cp to $y86$cp-pre.

Why? Because we need a stronger invariant, which we call $y86$cp, on the Y86 state than the stobj recognizer.

So what is this invariant?

```
(defun y86$cp (y86$c)
  (declare (xargs :stobjs y86$c))
  (and (y86$cp-pre y86$c)
       (good-memp y86$c)))
```
processor modeling: invariant on the processor state

we have renamed the recognizer for the \textit{y86$\text{c}$} stobj from \textit{y86$\text{cp}$} to \textit{y86$\text{cp-}\text{pre}$}.

why? because we need a stronger invariant, which we call \textit{y86$\text{cp}$}, on the y86 state than the stobj recognizer.

so what is this invariant?

\begin{verbatim}
(defun y86$\text{cp}$ (y86$c)
 (declare (xargs :stobjs y86$c))
 (and (y86$\text{cp-}\text{pre}$ y86$c)
      (good-\text{memp} y86$c))
\end{verbatim}
We have renamed the recognizer for the \texttt{y86\$c} stobj from \texttt{y86\$cp} to \texttt{y86\$cp-pre}.

Why? Because we need a stronger invariant, which we call \texttt{y86\$cp}, on the Y86 state than the stobj recognizer.

So what is this invariant?

\begin{verbatim}
(defun y86\$cp (y86\$c)
  (declare (xargs :stobjs y86\$c))
  (and (y86\$cp-pre y86\$c)
       (good-memp y86\$c)))
\end{verbatim}
Processor Modeling: \textit{mem$ci} and \textit{!mem$ci}

We define functions to read (\textit{mem$ci}) and write (\textit{!mem$ci}) to the memory.

\begin{verbatim}
(defun mem$ci (i y86$c)  
  (declare (xargs :stobjs y86$c  
                   :guard (and (integerp i)  
                               (<= 0 i)  
                               (< i *mem-size-in-bytes*)  
                               (y86$cp y86$c))))
  (let* ((i-top (ash i (- (* 2^x-byte-pseudo-page*)  
                      (y86$cp y86$c))))  
          (addr (mem-tablei i-top y86$c)))
    (cond ((eql addr 1) ;; page is not present  
           *default-mem-value*)
          (t (let ((index (logior addr (logand 16777215 i))))
              (mem-arrayi index y86$c)))))))
\end{verbatim}

We can define the usual read-over-write and write-over-write theorems about the memory using these two functions.
We define functions to read ($mem\text{\$}ci$) and write ($!mem\text{\$}ci$) to the memory.

```lisp
(defun mem$ci (i y86$c)
  (declare (xargs :stobjs y86$c
                   :guard (and (integerp i)
                               (<= 0 i)
                               (< i (*mem-size-in-bytes*)
                               (y86$cp y86$c))))
  (let* ((i-top (ash i (- (* 2^x-byte-pseudo-page*)
                      (y86$cp y86$c)))))
    (let ((addr (mem-tablei i-top y86$c)))
      (cond ((eql addr 1) ;; page is not present
              *default-mem-value*)
            (t (let ((index (logior addr (logand 16777215 i))))
                (mem-arrayi index y86$c))))))
```

We can define the usual read-over-write and write-over-write theorems about the memory using these two functions.
We define functions to read ($\text{mem}$$\text{ci}$) and write ($\text{!mem}$$\text{ci}$) to the memory.

(defun $\text{mem}$$\text{ci}$ (i y86$c)
  (declare (xargs :stobjs y86$c
                  :guard (and (integerp i)
                              (<= 0 i)
                              (< i (*mem-size-in-bytes*)
                                  (y86$cp y86$c))))
  (let* (((i-top (ash i (- (* 2^x-byte-pseudo-page*)
                     (y86$cp y86$c)))))
         (addr (mem-tablei i-top y86$c)))
    (cond ((eql addr 1) ;; page is not present
           *default-mem-value*)
          (t (let (((index (logior addr (logand 16777215 i)))
                    (mem-arrayi index y86$c))))))

We can define the usual read-over-write and write-over-write theorems about the memory using these two functions.
Processor Modeling: Memory Read-Write Theorem

Read-over-Write theorem:

(defthm read-write
  (implies (and (y86$cp y86$c)
                (integerp i)
                (<= 0 i)
                (< i *mem-size-in-bytes*)
                (integerp j)
                (<= 0 j)
                (< j *mem-size-in-bytes*)
                (n08p v))
  (equal (mem$ci j (!mem$ci i v y86$c))
         (if (equal i j)
             v
             (mem$ci j y86$c))))
We define a “classic” ACL2 instruction interpreter. Here’s the run function of the Y86:

```
(defund y86 (y86$c n)
  (declare (xargs :guard (and (natp n)
                              (y86$cp y86$c))
                     :measure (acl2-count n)
                     :stobjs (y86$c)))
  (if (mbe :logic (zp n) :exec (= n 0))
      y86$c
      (if (ms y86$c)
          y86$c
          (let ((y86$c (y86-step y86$c)))
            (y86 y86$c (1- n))))))
```
Processor Modeling: Y86 step function

Here's the step function:

(defun y86-step (y86$c)
  (declare (xargs :guard (y86$cp y86$c)
                   :stobjs (y86$c)))
  (b* ((pc (eip y86$c))
       (byte-at-pc (rm08 pc y86$c)))
       (case byte-at-pc

         ;; halt:  Stop the machine
         (#x00 (y86-halt y86$c))
         ...

         ;; jmp, jle, jl, je, jne, jge, jg:  Conditional jump
         (#x70 (y86-cjump y86$c 0))
         ...
         (#x76 (y86-cjump y86$c 6))
         ...
         (t (y86-illegal-opcode y86$c))))
)
Outline

Introduction

Processor Modeling

Abstract Stobjs

Two Advantages of Abstract Stobjs
   Eliminating Hypotheses
   Avoiding Expensive Guard-Checking

Proof by Symbolic Execution (GL)
   GL: Introduction
   Proof of Correctness of the Y86 Popcount Program

Conclusion
Abstract Stobjs: Introduction

Abstract Stobjs were introduced in ACL2 Version 5.0.

Abstract: \( a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \)

correspondence: \( \uparrow \uparrow \uparrow \ldots \)

Concrete: \( c_0 \rightarrow c_1 \rightarrow c_2 \rightarrow \ldots \)
Abstract Stobjs: defabsstobj

(defabsstobj y86
  :concrete y86$c
  :recognizer (y86p :logic y86$ap
    :exec y86$cp-pre)
  :creator (create-y86 :logic create-y86$a
    :exec create-y86$c)
  :corr-fn corr
  :exports ((eip :logic eip$a :exec eip$c)
    (!eip :logic !eip$a :exec !eip$c)
    ...
    (memi :logic mem$ai :exec mem$ci)
    (!memi :logic !mem$ai :exec !mem$ci)))
(defun-sk corr-mem (y86$c abs-memory-field)
  (forall i
   (implies (and (natp i)
                  (< i (*mem-size-in-bytes*))
                  (equal (mem$ci i y86$c)
                         (g i i abs-memory-field)))))

(defun-nx corr (c a)
  (and (y86$cp c)
       (y86$ap a)
       (equal (nth *eip* c) (nth *eip* a))
       ...
       (corr-mem c (nth *memi* a)))))
Abstract Stobj: Y86 Correspondence and Preservation Theorems

(defthm !memi{correspondence}
  (implies (and (corr y86$c y86)
                 (y86$ap y86)
                 (n32p i)
                 (n08p v))
   (corr (!mem$ci i v y86$c)
          (!mem$ai i v y86))))

(defthm !memi{preserved}
  (implies (and (y86$ap y86)
                 (n32p i)
                 (n08p v))
   (y86$ap (!mem$ai i v y86))))
Abstract Stobjs: Summary

Here are the steps involved in introducing a $\textit{defabsstobj}$ event.

- Define a “normal” stobj — $y86$c — using the $\textit{defstobj}$ event; we will call this the con$\textit{crete}$ stobj.
- Define the complicated invariant on the concrete stobj (i.e. $y86$cp).
- Define the function that describes how the concrete and abstract stobj will correspond. It is this function that appears in the proof obligations that must be met before a $\textit{defabsstobj}$ event is admitted.
- Define the accessors, updaters, and recognizers for the fields of the abstract stobj (and the creator function of the stobj).
- Prove the correspondence, preservation, and guard theorems for the above functions.
- Define the abstract stobj — $y86$ — corresponding to the concrete stobj.
Abstract Stobjs: Summary

Here are the steps involved in introducing a \textit{defabsstobj} event.

\begin{itemize}
\item Define a “normal” stobj — \texttt{y86\textasciicircum{}c} — using the \textit{defstobj} event; we will call this the \textit{concrete} stobj.
\item Define the complicated invariant on the concrete stobj (i.e. \texttt{y86\textasciicircum{}cp}).
\item Define the function that describes how the concrete and abstract stobj will correspond. It is this function that appears in the proof obligations that must be met before a \textit{defabsstobj} event is admitted.
\item Define the accessors, updaters, and recognizers for the fields of the abstract stobj (and the creator function of the stobj).
\item Prove the correspondence, preservation, and guard theorems for the above functions.
\item Define the abstract stobj — \texttt{y86} — corresponding to the concrete stobj.
\end{itemize}
Abstract Stobjs: Summary

Here are the steps involved in introducing a `defabsstobj` event.

- Define a “normal” stobj — `y86$c` — using the `defstobj` event; we will call this the `concrete` stobj.
- Define the complicated invariant on the concrete stobj (i.e. `y86$cp`).
- Define the function that describes how the concrete and abstract stobj will correspond. It is this function that appears in the proof obligations that must be met before a `defabsstobj` event is admitted.
- Define the accessors, updaters, and recognizers for the fields of the abstract stobj (and the creator function of the stobj).
- Prove the correspondence, preservation, and guard theorems for the above functions.
- Define the abstract stobj — `y86` — corresponding to the concrete stobj.
Abstract Stobjs: Summary

Here are the steps involved in introducing a \textit{defabsstobj} event.

- Define a “normal” stobj — $y86c$ — using the \textit{defstobj} event; we will call this the \textit{concrete} stobj.
- Define the complicated invariant on the concrete stobj (i.e. $y86cp$).
- Define the function that describes how the concrete and abstract stobj will correspond. It is this function that appears in the proof obligations that must be met before a \textit{defabsstobj} event is admitted.
- Define the accessors, updaters, and recognizers for the fields of the abstract stobj (and the creator function of the stobj).
- Prove the correspondence, preservation, and guard theorems for the above functions.
- Define the abstract stobj — $y86$ — corresponding to the concrete stobj.
Abstract Stobjs: Summary

Here are the steps involved in introducing a `defabsstobj` event.

- Define a “normal” stobj — \( y86c \) — using the `defstobj` event; we will call this the concrete stobj.
- Define the complicated invariant on the concrete stobj (i.e. \( y86cp \)).
- Define the function that describes how the concrete and abstract stobj will correspond. It is this function that appears in the proof obligations that must be met before a `defabsstobj` event is admitted.
- Define the accessors, updaters, and recognizers for the fields of the abstract stobj (and the creator function of the stobj).
- Prove the correspondence, preservation, and guard theorems for the above functions.
- Define the abstract stobj — \( y86 \) — corresponding to the concrete stobj.
Abstract Stobjs: Summary

Here are the steps involved in introducing a `defabsstobj` event.

- Define a “normal” stobj — `y86$c` — using the `defstobj` event; we will call this the `concrete` stobj.
- Define the complicated invariant on the concrete stobj (i.e. `y86$cp`).
- Define the function that describes how the concrete and abstract stobj will correspond. It is this function that appears in the proof obligations that must be met before a `defabsstobj` event is admitted.
- Define the accessors, updaters, and recognizers for the fields of the abstract stobj (and the creator function of the stobj).
- Prove the correspondence, preservation, and guard theorems for the above functions.
- Define the abstract stobj — `y86` — corresponding to the concrete stobj.
Outline

Introduction

Processor Modeling

Abstract Stobjs

Two Advantages of Abstract Stobjs
  Eliminating Hypotheses
  Avoiding Expensive Guard-Checking

Proof by Symbolic Execution (GL)
  GL: Introduction
  Proof of Correctness of the Y86 Popcount Program

Conclusion
Eliminating Hypotheses

(defthm read-write
  ;; NO hypotheses at all!
  (equal (memi i (!memi j v y86))
    (if (equal i j)
        (or v 0)
        (memi i y86))))

Compare this theorem with the read-write theorem we saw earlier!
Avoiding Expensive Guard Checking

(defund y86 (y86 n)
   (declare (xargs :guard (natp n)
                   :measure (acl2-count n)
                   :stobjs (y86)))
   (if (mbe :logic (zp n) :exec (= n 0))
       y86
       (if (ms y86)
           y86
           (let ((y86 (y86-step y86)))
             (y86 y86 (1- n))))))

Compare this function with the run function we saw earlier!

ACL2 does not evaluate calls to the stobj recognizer!
Avoiding Expensive Guard Checking

(defund y86 (y86 n)
  (declare (xargs :guard (natp n)
                  :measure (acl2-count n)
                  :stobjs (y86)))
  (if (mbe :logic (zp n) :exec (= n 0))
      y86
    (if (ms y86)
        y86
      (let ((y86 (y86-step y86)))
        (y86 y86 (1- n)))))))

Compare this function with the run function we saw earlier!

ACL2 does not evaluate calls to the stobj recognizer!
Avoiding Expensive Guard Checking

(defund y86 (y86 n)
  (declare (xargs :guard (natp n)
                  :measure (acl2-count n)
                  :stobjs (y86)))
  (if (mbe :logic (zp n) :exec (= n 0))
    y86
    (if (ms y86)
      y86
      (let ((y86 (y86-step y86)))
        (y86 y86 (1- n))))))

Compare this function with the run function we saw earlier!

ACL2 does not evaluate calls to the stobj recognizer!
Outline

Introduction

Processor Modeling

Abstract Stobjs

Two Advantages of Abstract Stobjs
  Eliminating Hypotheses
  Avoiding Expensive Guard-Checking

Proof by Symbolic Execution (GL)
  GL: Introduction
  Proof of Correctness of the Y86 Popcount Program

Conclusion
GL and Symbolic Execution

- GL is a framework for proving *finite* ACL2 theorems.
- GL can *symbolically execute* finite terms.
Popcount Program in the Y86

<Demo>
An Observation...

- GL symbolically executes functions according to ACL2 logic...

- Which means: GL symbolically executes the logical definitions of the stobj functions (like \textit{rgfi} in our popcount proof) — we do not get the performance of stobj operations.
An Observation...

- GL symbolically executes functions according to ACL2 logic...

- Which means: GL symbolically executes the logical definitions of the stobj functions (like \textit{rgfi} in our popcount proof) — we do not get the performance of stobj operations.
How did abstract stobjs help?

- Imagine we only had the concrete stobj.
  - Inconvenient to do symbolic execution with the logical representation of a large stobj memory
  - GL would need to symbolically execute $\text{mem}^{ci}$ and $\neg\text{mem}^{ci}$, which have complicated definitions

- With abstract stobjs, we have:
  - A smaller memory representation since we use records to model the memory
  - Simpler definitions of $\text{memi}$ and $\neg\text{memi}$
How did abstract stobjs help?

- Imagine we only had the concrete stobj.
  - Inconvenient to do symbolic execution with the logical representation of a large stobj memory
  - GL would need to symbolically execute \textit{mem}^\textit{ci} and \textit{!mem}^\textit{ci}, which have complicated definitions

- With abstract stobjs, we have:
  - A smaller memory representation since we use records to model the memory
  - Simpler definitions of \textit{memi} and \textit{!memi}
How did abstract stobjs help?

▶ Imagine we only had the concrete stobj.
  ▶ Inconvenient to do symbolic execution with the logical representation of a large stobj memory
  ▶ GL would need to symbolically execute $mem$ and $!mem$, which have complicated definitions

▶ With abstract stobjs, we have:
  ▶ A smaller memory representation since we use records to model the memory
  ▶ Simpler definitions of $mem$ and $!mem$
How did abstract stobjs help?

- Imagine we only had the concrete stobj.
  - Inconvenient to do symbolic execution with the logical representation of a large stobj memory
  - GL would need to symbolically execute `mem$ci` and `!mem$ci`, which have complicated definitions

- With abstract stobjs, we have:
  - A smaller memory representation since we use records to model the memory
  - Simpler definitions of `memi` and `!memi`
How did abstract stobjs help?

- Imagine we only had the concrete stobj.
  - Inconvenient to do symbolic execution with the logical representation of a large stobj memory
  - GL would need to symbolically execute `mem$ci` and `!mem$ci`, which have complicated definitions

- With abstract stobjs, we have:
  - A smaller memory representation since we use records to model the memory
  - Simpler definitions of `memi` and `!memi`
How did abstract stobjs help?

- Imagine we only had the concrete stobj.
  - Inconvenient to do symbolic execution with the logical representation of a large stobj memory
  - GL would need to symbolically execute $\text{mem$ci}$ and $!\text{mem$ci}$, which have complicated definitions

- With abstract stobjs, we have:
  - A smaller memory representation since we use records to model the memory
  - Simpler definitions of $\text{memi}$ and $!\text{memi}$
Outline

Introduction

Processor Modeling

Abstract Stobjs

Two Advantages of Abstract Stobjs
  Eliminating Hypotheses
  Avoiding Expensive Guard-Checking

Proof by Symbolic Execution (GL)
  GL: Introduction
  Proof of Correctness of the Y86 Popcount Program

Conclusion
Conclusion

We talked about:

- How, during our work on processor models, we realized the need for abstract stobjs
- How abstract stobjs solved our problems and made it possible to:
  - Prove theorems with fewer hypotheses
  - Avoid expensive guard checking
  - Use GL to do proofs involving large stobj memories
Conclusion

We talked about:

► How, during our work on processor models, we realized the need for abstract stobjs

► How abstract stobjs solved our problems and made it possible to:
  ► Prove theorems with fewer hypotheses
  ► Avoid expensive guard checking
  ► Use GL to do proofs involving large stobj memories
Thank You!