An Abstract, Approximation-Based Approach to Embedded Code Pointers and Partial-Correctness

Zhaozhong Ni

Microsoft Research
One Microsoft Way, Redmond, WA 98052, U.S.A.
zhaozhong.ni@microsoft.com

Abstract

To support higher-order type-like features such as embedded code pointers in logic-based verification, one approach is to build a syntactic assertion logic that combines logic and types. But it is not totally satisfactory in various aspects. Another approach is to use approximation to simulate the behavior of types and typing invariants in logic, but this pollutes program specifications and proofs with complex approximation details. Additionally, existing approximation-based work supports embedded code pointers without partial-correctness guarantee.

We propose a new abstract, approximation-based approach to support embedded code pointers in logic-based verification. Our specification language and inference rules are independent of approximation, thus allowing programs to be certified abstractly. For the support of embedded code pointers, this benefits not only interactive-verification, but certifying compilation and automated theorem proving as well. Approximation is only used to establish soundness and partial-correctness. This proves to be advantageous for meta theory design and mechanizing. Additionally, we easily support dynamic code generation. The central idea should be applicable to other higher-order features. Our work is presented on and mechanized in, but not limited to, assembly languages and the Coq proof-assistant.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

Embedded code pointers (ECPs, e.g., higher-order functions), among other features such as general reference and recursive types, are common features found in type systems, such as typed assembly languages (TAL) [11, 6]. In the context of proof-carrying code [14, 1], there have been more focuses on certifying systems and low-level code that are hard to handle safely with traditional type systems, as well as reducing the size of trusted computing base (TCB) to the minimum. Recent work such as [1, 2, 3, 4, 16, 18, 13, 12, 8] aim to combine the expressive power of Hoare logic [7, 9] with the modularity of type systems to achieve better logic-based verification. It is important for these systems to support those higher-order features well, particularly for ECPs, which are crucial for objects, closures, and modularity in general.

Depending on the expressiveness and productivity requirements, different components of software system are and will be certified at different abstraction levels (e.g., high-level, intermediate, and assembly) by different methods (e.g., type-based, automated first-order logic based, and interactive high-order logic based). Unlike type systems which are userfriendly for the support of ECPs (the reasoning of syntactic ECP types is not done by the programmer, thus the complexity of ECP support details is irrelevant provided they are sound), logic-based verifications present user friendliness as an important issue: both theorem prover writers/users and interactive-verification programmer need to be able to comfortably deal with ECP formulas and their reasoning, on top of the already daunting task of first/higher-logic reasoning. Since the control flows which integrate software components are often higher-order and involve ECPs, to obtain guarantees about multiple components or about the entire software system, certifying compiler writers desire not only user friendliness, but also good specification/proof portability for the support of ECPs.

One approach to support embedded code pointers and other higher-order features is to build new "syntactic" assertion logic layers, combining higher- or first-order logic and type constructors, above the (mechanized) meta logic. Recent examples of these approaches include Spec# by Barnett et al. [4], XCAP by Ni and Shao [16, 18, 17], HTT by Nanevski et al. [13, 12], and GTAL by Hawblitzel et al. [8]. Since these systems are designed with type systems being the backbone, their support of higher-order features is similar to that of traditional type systems in both modularity and user friendliness. Some of them have been used to do interactive [18, 8] or automated [4] program verification, as well as source [4] or target [17, 8] languages for certifying compilation. They can also guarantee partial-correctness (implicit in some cases). However, there are several aspects in which these work are not totally satisfactory, especially for interactive and automated logic-based program verification and proof-preserving certifying compilation.

One problem is the establishment and mechanization of the meta theory. Since the hybrid assertion logic layers are quite expressive, the corresponding meta theories are also large and complex, often more complex than a typical higher-order logic/type system. Formalizing their meta properties, such as consistency and soundness, is by no means trivial. As a matter of fact, XCAP is the only one of the above systems that comes with fully mechanized meta theory. Most of the other systems come only with meta theory

on paper for subsets of their frameworks. This is clearly not ideal for logic-based verification.

Since the assertion and meta logics are not at the same level, in assertion logic it is not possible to directly express and reason about meta properties related to those type-like features. For example, to permit dynamic allocation, current program specification has to be "monotonic" over the growth of the heap. For syntactic type systems, this may hold true for all possible types (e.g., in TAL there is usually a "heap extension" lemma [11]). In hybrid assertion logics, one has to carefully control the expressive power of program specifications or provide special "stubs" to indirectly refer to these properties, both bringing in more complexity.

Similarly, as those type-like features usually require the creation and maintenance of certain meta-level invariants, such as global heap typing, it is difficult to reason about runtime operations involving or temporarily breaking these invariants. Examples of these are self-modifying code, garbage collections, concurrency runtimes, etc. In fact, in [8, 10], two different assertion logics have to be used for the mutator and collector, respectively, to certify systems with GC. This introduces additional complexity into the system and enlarges the TCB.

An alternative approach to support embedded code pointers and other type-like features is by doing "semantic" approximation in logic to simulate the behavior of type constructors and typing invariants. Among recent work in this line, the most representative one is the FPCC semantic model work by Appel et al. [1, 2, 3]. Since there is no additional syntactic layers defined, there is no need to establish large and complex meta theories. Because the semantic approximation model of those higher-order features are defined in the same logic in which the latter is defined, one could potentially specify and reason about "meta" properties of the program specifications, and even certify those run-time code hidden below the type-interface, such as dynamic code generation and garbage collection.

Nevertheless, there are also drawbacks that are currently preventing the approximation-based approach from being used effectively for logic-based verification. One problem is the pollution of program specifications and proofs with complex approximation model details. Not only are they not user-friendly (not too surprising given that these systems [1, 2, 3] are primarily designed for foundational proof generation from a typial typed-assembly language, albeit making it challenging for them to be used for logic-based verifications purpose), they also prevent program specifications and proofs from being reusable across different approximation (or potentially non-approximation) models (for different feature sets). In addition, existing approximation-based work have been focusing on non-stuckness and can only support embedded code pointers without partial-correctness guarantee [3].

In this paper, we propose a new approach to embedded code pointers and partial-correctness in logic-based verification. We follow the approximation-based idea and apply it to embedded code pointers, making sure partial-correctness is not sacrificed. Although our approximation model is already constructed in a relatively lightweight and modular way, we do not stop here. We design an abstract specification language, a.k.a, assertion logic, as well as a set of abstract program inference rules, a.k.a., program logic. Both our assertion and program logics are completely **abstract** from any approximation details. Users only need to learn and manipulate these "approximation-free" constructs (about 110 lines of Coq code) to certify, compile, and link programs with ECPs while enjoying partial-correctness guarantee and much better productivity than the syntactic approach.

The approximation model we built for embedded code pointers is only used to establish the soundness and partial-correctness theorems (the definition of which are also "approximation-free"). Our approximation is expressive enough to express dynamic code generations and other forms of code morphing. By avoiding building syntactic assertion logic, we avoided building complex meta theories and being hindered by various implementation annoyances. The central idea of this paper should be applicable to other higher-order features, such as general references and recursive types. Our work is presented on an ideal RISC assembly machine and fully mechanized in the Coq proof-assistant. However, our result should apply to other abstract levels as well as similar higher-order logics. Our full mechanization result is available for download at [15].

The paper is organized as follows. We first introduce our target machine and briefly review the syntactic XCAP approach [16] and semantic approximation-based FPCC approach [3] in Sect. 2. We then present the user-friendly abstract assertion logic and program logic in Sect. 3. In Sect. 4, we present our approximation-based model for embedded code pointers and prove soundness and partial-correctness. We show how to support dynamic code generation in Sect. 5. Finally, we compare with related work and conclude in Sect. 6.

2. Background

We use Coq as a mechanized meta logic and assume the following syntax.

$$\begin{aligned}(\textit{Term})\ A, B &::= \mathsf{Type} \mid x \mid \lambda x{:}A.\, B \mid A\ B \mid A \to B \\ &\quad \mid \Pi x{:}A.\, B \mid \mathit{inductive\ definitions} \mid \ldots \\ (\textit{Prop})\ p, q &::= \mathsf{True} \mid \mathsf{False} \mid p \wedge q \mid p \vee q \mid p \supset q \mid \forall x{:}A.\, p \\ &\quad \mid \exists x{:}A.\, p \mid \mathit{inductive\ predicates} \mid \ldots\end{aligned}$$

For non-dependent and dependent pairs, we use $A \times B$ and $\Sigma x : A.\, B$ to represent their types, and (A, B) and $\langle A, B\rangle$ for their terms. We also write $\Sigma x.\, B$ for $\Sigma x{:}\mathsf{Type}.\, B$.

Target machine. We use a common target machine throughout our discussions. We define the syntax and the operational semantics of the target machine in Figure 1. A complete program 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 is minimal but its extensions are straightforward. The operational semantics of this language (see Figure 2) should pose no surprise. Note that it is illegal to access undefined heap locations, or jump to a code label that does not exist; under both cases, the execution gets stuck.

Syntactic approach: XCAP. XCAP [16] is a logic-based verification framework for assembly code with modular support of embedded code pointers and impredicative polymorphisms. It has been used to mechanically verify partial-correctness of an x86 machine context library [18] in Coq. XCAP's assertion logic works by defining a syntactic layer called *extended logical propositions* ($PropX$) above the meta logic and using syntactic inference rules to establish the validity of $PropX$.

Figure 3 defines $PropX$, the core of the XCAP specification. *PropX* can be viewed as a lift of the (Coq) meta logic propositions, extended with a `cptr` constructor. Roughly speaking, (`cptr f a`) asserts that code label `f` is valid with precondition `a`. $PropX$ can be used to construct assertions. For example, to specify that `r1` and `r2` store different values, we write $\lambda(\mathbb{H}, \mathbb{R}).\ \langle \mathbb{R}(\mathtt{r1}) \neq \mathbb{R}(\mathtt{r2})\rangle$. We define code heap specifications and assertion subsumptions accordingly.

To establish the *validity of extended propositions*, XCAP defines a set of syntactic validity rules. The interpretation of extended propositions $[\![\, \mathtt{P} \,]\!]_\Psi$ is defined as their validity under the empty

$$
\begin{array}{rrl}
(\textit{Program})\ \mathbb{P} & ::= & (\mathbb{C}, \mathbb{S}, \mathbb{I}) \\
(\textit{State})\ \mathbb{S} & ::= & (\mathbb{H}, \mathbb{R}) \\
(\textit{Mem})\ \mathbb{H} & ::= & \{\mathtt{l} \rightsquigarrow \mathtt{w}\}^* \\
(\textit{Regfile})\ \mathbb{R} & ::= & \{\mathtt{r} \rightsquigarrow \mathtt{w}\}^* \\
(\textit{Reg})\ \mathtt{r} & ::= & \{\mathtt{rk}\}^{\mathtt{k} \in \{0 \ldots 31\}} \\
(\textit{Word}, \textit{Labels})\ \mathtt{w}, \mathtt{f}, \mathtt{l} & ::= & i \ \ (\textit{nat nums}) \\
(\textit{CodeHeap})\ \mathbb{C} & ::= & \{\mathtt{f} \rightsquigarrow \mathbb{I}\}^* \\
(\textit{InstrSeq})\ \mathbb{I} & ::= & \mathtt{c}; \mathbb{I} \mid \mathsf{jd}\ \mathtt{f} \mid \mathsf{jmp}\ \mathtt{r} \\
(\textit{Instr})\ \mathtt{c} & ::= & \mathsf{bgti}\ \mathtt{r}_s, i, \mathtt{f} \mid \mathsf{addi}\ \mathtt{r}_d, \mathtt{r}_s, i \\
 & & \mid \mathsf{add}\ \mathtt{r}_d, \mathtt{r}_s \mathtt{r}_t \mid \mathsf{movi}\ \mathtt{r}_d, i \\
 & & \mid \mathsf{mov}\ \mathtt{r}_d, \mathtt{r}_s \mid \mathsf{ld}\ \mathtt{r}_d, \mathtt{r}_s(i) \\
 & & \mid \mathsf{st}\ \mathtt{r}_d(i), \mathtt{r}_s \mid \ldots
\end{array}
$$

Figure 1. Selected syntax of target machine

if $\mathbb{I} =$	then $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}) \longmapsto$
$\mathsf{jd}\ \mathtt{f}$	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{C}(\mathtt{f}))$ when $\mathtt{f} \in \mathsf{dom}(\mathbb{C})$
$\mathsf{jmp}\ \mathtt{r}$	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{C}(\mathbb{R}(\mathtt{r})))$ when $\mathbb{R}(\mathtt{r}) \in \mathsf{dom}(\mathbb{C})$
$\mathsf{bgti}\ \mathtt{r}_s, i, \mathtt{f}; \mathbb{I}'$	$(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}')$ when $\mathbb{R}(\mathtt{r}_s) \leq i$; $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{C}(\mathtt{f}))$ when $\mathbb{R}(\mathtt{r}_s) > i$
$\mathtt{c}; \mathbb{I}'$	$(\mathbb{C}, \mathtt{Next_c}(\mathbb{H}, \mathbb{R}), \mathbb{I}')$

if $\mathtt{c} =$	then $\mathtt{Next_c}(\mathbb{H}, \mathbb{R}) =$
$\mathsf{add}\ \mathtt{r}_d, \mathtt{r}_s \mathtt{r}_t$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s) + \mathbb{R}(\mathtt{r}_t)\})$
$\mathsf{mov}\ \mathtt{r}_d, \mathtt{r}_s$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{R}(\mathtt{r}_s)\})$
$\mathsf{movi}\ \mathtt{r}_d, i$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow i\})$
$\mathsf{ld}\ \mathtt{r}_d, \mathtt{r}_s(i)$	$(\mathbb{H}, \mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{H}(\mathbb{R}(\mathtt{r}_s) + i)\})$ when $\mathbb{R}(\mathtt{r}_s) + i \in \mathsf{dom}(\mathbb{H})$
$\mathsf{st}\ \mathtt{r}_d(i), \mathtt{r}_s$	$(\mathbb{H}\{\mathbb{R}(\mathtt{r}_d) + i \rightsquigarrow \mathbb{R}(\mathtt{r}_s)\}, \mathbb{R})$ when $\mathbb{R}(\mathtt{r}_d) + i \in \mathsf{dom}(\mathbb{H})$

Figure 2. Selected operational semantics of target machine

environment. Consistency of $PropX$ interpretation, "$[\![\langle \mathsf{False} \rangle]\!]_\Psi$ is not provable," is a corollary of the following theorem.

THEOREM 2.1 (Soundness of $PropX$ Interpretation).

- If $[\![\langle p \rangle]\!]_\Psi$ then p;
- if $[\![\mathsf{cptr}\ \mathtt{f}\ \mathtt{a}]\!]_\Psi$ then $\Psi(\mathtt{f}) = \mathtt{a}$;
- if $[\![\mathtt{P} \mathbin{\mathbb{\wedge}} \mathtt{Q}]\!]_\Psi$ then $[\![\mathtt{P}]\!]_\Psi$ and $[\![\mathtt{Q}]\!]_\Psi$;
- if $[\![\mathtt{P} \mathbin{\mathbb{\vee}} \mathtt{Q}]\!]_\Psi$ then either $[\![\mathtt{P}]\!]_\Psi$ or $[\![\mathtt{Q}]\!]_\Psi$;
- if $[\![\mathtt{P} \mathbin{\mathbb{\rightarrow}} \mathtt{Q}]\!]_\Psi$ and $[\![\mathtt{P}]\!]_\Psi$ then $[\![\mathtt{Q}]\!]_\Psi$;
- if $[\![\forall \alpha : A.\, \mathtt{P}]\!]_\Psi$ and $B : A$ then $[\![\mathtt{P}[B/\alpha]]\!]_\Psi$;
- if $[\![\exists \alpha : A.\, \mathtt{P}]\!]_\Psi$ then there exists $B : A$ such that $[\![\mathtt{P}[B/\alpha]]\!]_\Psi$;
- if $[\![\forall \alpha : A \rightarrow \textit{PropX}.\, \mathtt{P}]\!]_\Psi$ and $\mathtt{a} : A \rightarrow \textit{PropX}$ then $[\![\mathtt{P}[\mathtt{a}/\alpha]]\!]_\Psi$;
- if $[\![\exists \alpha : A \rightarrow \textit{PropX}.\, \mathtt{P}]\!]_\Psi$ then there exists $\mathtt{a} : A \rightarrow \textit{PropX}$ such that $[\![\mathtt{P}[\mathtt{a}/\alpha]]\!]_\Psi \ldots$.

On the program logic side, XCAP shares great similarity with typical typed assembly languages. We present the XCAP inference rules in Figure 4. A program is well-formed in XCAP if each of its components is. For a code heap to be well-formed, each block in it must be well-formed. The intuition behind well-formed instruction sequence judgment is that if the state satisfies precondition $\mathtt{a}$, then executing instruction sequence $\mathbb{I}$ is safe with respect to Ψ. For simple instructions, XCAP requires the pre- and post-condition to satisfy the weakest precondition relation. A direct jump is safe as long as the target code's precondition is weaker than the current one. For an indirect jump, the current precondition has to guarantee that the target address is a well-formed code label with a weaker precondition. The (ECP) rule allows one to make local code pointers first-class.

$$
\begin{array}{rrcl}
(\textit{PropX})\ & \mathtt{P}, \mathtt{Q} & ::= & \langle p \rangle \mid \mathsf{cptr}\ \mathtt{f}\ \mathtt{a} \mid \mathtt{P} \mathbin{\mathbb{\wedge}} \mathtt{Q} \mid \mathtt{P} \mathbin{\mathbb{\vee}} \mathtt{Q} \mid \mathtt{P} \mathbin{\mathbb{\rightarrow}} \mathtt{Q} \\
 & & & \mid \forall x : A.\, \mathtt{P} \mid \exists x : A.\, \mathtt{P} \\
 & & & \mid \forall \alpha : A \rightarrow \textit{PropX}.\, \mathtt{P} \\
 & & & \mid \exists \alpha : A \rightarrow \textit{PropX}.\, \mathtt{P} \\
(\textit{Assertion}) & \mathtt{a} & \in & \textit{State} \rightarrow \textit{PropX} \\
(\textit{CdHpSpec}) & \Psi & ::= & \{\mathtt{f} \rightsquigarrow \mathtt{a}\}^* \\
(\textit{AssertImp}) & \mathtt{a} \Rightarrow \mathtt{a}' & \triangleq & \forall \Psi, \mathbb{S}.\, [\![\mathtt{a}]\!]_\Psi\, \mathbb{S} \supset [\![\mathtt{a}']\!]_\Psi\, \mathbb{S} \\
(\textit{StepImp}) & \mathtt{a} \Rightarrow_{\mathtt{c}} \mathtt{a}' & \triangleq & \forall \Psi, \mathbb{S}.\, [\![\mathtt{a}]\!]_\Psi\, \mathbb{S} \supset [\![\mathtt{a}']\!]_\Psi\, \mathtt{Next_c}(\mathbb{S})
\end{array}
$$

Figure 3. Assertion language of XCAP

$\boxed{\Psi_G \vdash \{\mathtt{a}\}\, \mathbb{P}}$ **(*Well-formed Program*)**

$$\frac{\Psi_G \vdash \mathbb{C} : \Psi_G \qquad \Psi_G \vdash \{\mathtt{a}\}\, \mathbb{I} \qquad ([\![\mathtt{a}]\!]_{\Psi_G}\, \mathbb{S})}{\Psi_G \vdash \{\mathtt{a}\}\, (\mathbb{C}, \mathbb{S}, \mathbb{I})}\ (\textsc{prog})$$

$\boxed{\Psi_{IN} \vdash \mathbb{C} : \Psi}$ **(*Well-formed Code Heap*)**

$$\frac{\Psi_{IN} \vdash \{\mathtt{a}_i\}\, \mathbb{I}_i \qquad \forall \mathtt{f}_i}{\Psi_{IN} \vdash \{\mathtt{f}_1 \rightsquigarrow \mathbb{I}_1, \ldots, \mathtt{f}_n \rightsquigarrow \mathbb{I}_n\} : \{\mathtt{f}_1 \rightsquigarrow \mathtt{a}_1, \ldots, \mathtt{f}_n \rightsquigarrow \mathtt{a}_n\}}\ (\textsc{cdhp})$$

$\boxed{\Psi \vdash \{\mathtt{a}\}\, \mathbb{I}}$ **(*Well-formed Instruction Sequence*)**

$$\frac{\mathtt{a} \Rightarrow_{\mathtt{c}} \mathtt{a}' \qquad \Psi \vdash \{\mathtt{a}'\}\, \mathbb{I} \qquad \mathtt{c} \in \{\mathsf{add}, \mathsf{addi}, \mathsf{mov}, \mathsf{movi}, \mathsf{ld}, \mathsf{st}, \ldots\}}{\Psi \vdash \{\mathtt{a}\}\, \mathtt{c}; \mathbb{I}}\ (\textsc{seq})$$

$$\frac{\begin{array}{l}(\lambda(\mathbb{H}, \mathbb{R}).\, \langle \mathbb{R}(\mathtt{r}_s) \leq i \rangle \mathbin{\mathbb{\wedge}} \mathtt{a}\, (\mathbb{H}, \mathbb{R})) \Rightarrow \mathtt{a}' \\ (\lambda(\mathbb{H}, \mathbb{R}).\, \langle \mathbb{R}(\mathtt{r}_s) > i \rangle \mathbin{\mathbb{\wedge}} \mathtt{a}\, (\mathbb{H}, \mathbb{R})) \Rightarrow \Psi(\mathtt{f}) \\ \Psi \vdash \{\mathtt{a}'\}\, \mathbb{I} \qquad \mathtt{f} \in \mathsf{dom}(\Psi)\end{array}}{\Psi \vdash \{\mathtt{a}\}\, \mathsf{bgti}\ \mathtt{r}_s, i, \mathtt{f}; \mathbb{I}}\ (\textsc{bgti})$$

$$\frac{\mathtt{a} \Rightarrow \Psi(\mathtt{f}) \quad \mathtt{f} \in \mathsf{dom}(\Psi)}{\Psi \vdash \{\mathtt{a}\}\, \mathsf{jd}\ \mathtt{f}}\ (\textsc{jd})$$

$$\frac{\mathtt{a} \Rightarrow (\lambda(\mathbb{H}, \mathbb{R}).\, \exists \mathtt{a}'.\, \mathtt{a}'\, (\mathbb{H}, \mathbb{R}) \mathbin{\mathbb{\wedge}} \mathsf{cptr}\, \mathbb{R}(\mathtt{r})\, \mathtt{a}')}{\Psi \vdash \{\mathtt{a}\}\, \mathsf{jmp}\ \mathtt{r}}\ (\textsc{jmp})$$

$$\frac{(\lambda \mathbb{S}.\, \mathsf{cptr}\ \mathtt{f}\ \Psi(\mathtt{f}) \mathbin{\mathbb{\wedge}} \mathtt{a}\, \mathbb{S}) \Rightarrow \mathtt{a}' \quad \mathtt{f} \in \mathsf{dom}(\Psi) \quad \Psi \vdash \{\mathtt{a}'\}\, \mathbb{I}}{\Psi \vdash \{\mathtt{a}\}\, \mathbb{I}}\ (\textsc{ecp})$$

Figure 4. Inference rules of XCAP

The following derived rules allow weakening of preconditions, extension of code heap specifications, and safe static linking of separately certified code heaps.

$$\frac{\Psi \vdash \{\mathtt{a}\}\,\mathbb{I} \qquad \mathtt{a}' \Rightarrow \mathtt{a} \qquad \Psi' \supseteq \Psi}{\Psi' \vdash \{\mathtt{a}'\}\,\mathbb{I}}\ (\textsc{weaken})$$

$$\frac{\begin{array}{l}\Psi_{IN\,1} \vdash \mathbb{C}_1 : \Psi_1 \qquad \Psi_{IN\,2} \vdash \mathbb{C}_2 : \Psi_2 \\ \mathsf{dom}(\mathbb{C}_1) \cap \mathsf{dom}(\mathbb{C}_2) = \emptyset \\ \Psi_{IN\,1}(\mathtt{f}) = \Psi_{IN\,2}(\mathtt{f}) \qquad \forall \mathtt{f} \in \mathsf{dom}(\Psi_{IN\,1}) \cap \mathsf{dom}(\Psi_{IN\,2})\end{array}}{\Psi_{IN\,1} \cup \Psi_{IN\,2} \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi_1 \cup \Psi_2}\ (\textsc{link})$$

XCAP suffers from the same problems as other "syntactic" approaches. For example, although structure-wise $PropX$ is similar to first-order propositional logic, soundness and consistency of $PropX$ interpretation still take around 5,000 lines [18] of very dense Coq code to establish.

For another example, the following proposition says assertion a is monotonic over the extension of code heap:

$$\forall \mathbb{S}, \Psi, \Psi'. \Psi \subseteq \Psi' \supset [\![\, \mathtt{a}\, \mathbb{S} \,]\!]_{\Psi} \supset [\![\, \mathtt{a}\, \mathbb{S} \,]\!]_{\Psi'}$$

yet there is no general way to embed this meta-proposition into a program assertion. Thus, one can not expect to be able to extend code heap dynamically.

Approximation-based approach: FPCC / modal model. The modal model [3] uses approximation to build semantic models for typed assembly languages. With no syntactic assertion logic defined, there is no need to establish large complex meta theories. Because the semantic approximation model of those higher-order features are defined in the same logic in which the latter is defined and used, one could specify and reason about "meta" properties of the program specifications, such as "necessity" (i.e., monotonicity over "world" growth). For example, they have no problem in supporting dynamic allocation of general reference cells.

Instead of using the original notations and formulas from [3], we transform and present them in a way that matches better with the rest of this paper.

For example, instead of defining data heap specification as the intuitive yet ill-formed definition as follows ($\rightharpoonup$ stands for partial mapping)

$$dSpec \triangleq Label \rightharpoonup (Word \to dSpec \to Prop)$$

since $dSpec$ appears on both side of the definition and causes circularity, Appel et al. defined the following indexed code heap specification

$$dSpec\, n \triangleq Label \rightharpoonup (Word \to \bigcup_{i<n} dSpec\, i \to Prop)$$

where a collection of *weaker* data heap specification, $\bigcup_{i<n} dSpec\, i$, are used to approximate $dSpec$ up to $(n-1)$ steps in the future of execution. And the index-less data heap specification type is just a dependent pair:

$$DSpec \triangleq \Sigma n. \bigcup_{i<n} dSpec\, i.$$

The general reference predicate ref is then defined as

$$\begin{array}{l}\mathsf{ref}\ \mathtt{l}\ \mathtt{t}\ (\langle n, (\Psi_n, \ldots, \Psi_0)\rangle : DSpec) \\ \quad \triangleq \forall 0 < i <= n.\ \llcorner t \lrcorner_i \iff \Psi_i(\mathtt{l})\end{array}$$

where the value type t is of type $Word \to DSpec \to Prop$.

Appel et al. [3] supports embedded code pointers following the same approach by reusing the natural number index in the "world" for approximation. These approximation details have to appear in program specifications and proofs. Additionally, the definition of code pointer predicate in [3], presented as the following, prohibits partial-correctness. (We use the original annotations here.)

$$\begin{array}{rcl}\mathsf{codeptr}(\tau) & \triangleq & \exists \mathtt{l} : \mathsf{Loc} \wedge \triangleright !(\mathsf{slot}(\mathsf{pc}, \mathsf{just}\ \mathtt{l}) \wedge \tau \Rightarrow \mathsf{safe}). \\ \mathsf{safe} & \triangleq & \forall m : Mem.\ \mathsf{validmem}(m) \ \Rightarrow\ \mathsf{safemem}(m) \\ \mathsf{safemem}(m) & \triangleq & \{((n, \Psi), v) \mid safe_n(v, m)\} \\ safe_0(s) & \triangleq & \mathsf{True} \\ safe_{k+1}(s) & \triangleq & (\exists s'.\ s \mapsto s') \wedge \forall s'.\ s \mapsto s' \ \Rightarrow\ safe_k\ s'.\end{array}$$

For any indirect jump to an embedded code pointer specified with codeptr, after the jump, the only thing guaranteed is non-stuckness. This makes it much less interesting for logic-based verification where expressive power is desired.

3. User Friendly Abstract Assertion and Program Logics

In this section, we present an assertion logic (specification language) and a program logic (program inference rules). Both are defined without any knowledge of the details of approximation, or whether approximation is even used at all.

We notice that syntactic approaches such as XCAP, as well as syntactic type systems such as TAL, provide conceptually clean and user-friendly support of embedded code pointers. One key observation here is that embedded code pointer predicates and types are used and reasoned about by programmers in an abstract way. Take XCAP for example, when doing proof for well-formedness of code blocks, no possible reasoning about cptr constructor depends on the actual interpretation of it. In Figure 4, one can see that cptr propositions can be introduced by rule (ECP), consumed by rule (JMP), or they can get either preserved or dropped during subsumptions.

So the first idea here is to make this $cptr$ predicate as abstract as possible. The question is what is the type of $cptr$?

The parameters to $cptr$ are the address, the precondition, as well as the global code heap specification (conceptually, a mapping from addresses to preconditions). Since the global code heap specification is only used by $cptr$, we make the specification and its type abstract. Our assertion type now becomes

$$\begin{array}{l}\forall Spec : Type. \\ \forall cptr : Label \to (State \to Spec \to Prop) \to Spec \to Prop. \\ \forall \Psi : Spec. \\ \qquad State \to Prop.\end{array}$$

Since the three quantified variables above are used for the same purpose—embedded code pointers, we pack them and obtain the following type

$$\begin{array}{l}\Sigma Spec. \\ \quad (Label \to (State \to Spec \to Prop) \to Spec \to Prop) \times Spec\end{array}$$

which we name in Figure 5 as $\mathcal{X}$. After moving ahead the position of $State$ in the above assertion type, our assertion should now be of type

$$State \to \mathcal{X} \to Prop.$$

Comparing this with the XCAP assertion definition in Figure 3, we define $\mathcal{X} \to Prop$ as our new $PropX$ as in Figure 5, together with other constructs.

$$
\begin{array}{rrcl}
(\mathcal{X}) & \sigma & \in & \Sigma Spec.\,(Label \to \\
& & & \quad (State \to Spec \to Prop) \to \\
& & & \quad Spec \to Prop) \\
& & & \times Spec \\
(PropX) & \mathtt{P} & \in & \mathcal{X} \to Prop \\
(Assertion) & \mathtt{a} & \in & State \to PropX \\
(CdHpSpec) & \Psi & ::= & \{\mathtt{f} \rightsquigarrow \mathtt{a}\}^* \\
(AssertImp) & \mathtt{a} \Rightarrow \mathtt{a}' & \triangleq & \forall \sigma, \mathbb{S}.\, \mathtt{a}\ \mathbb{S}\ \sigma \supset \mathtt{a}'\ \mathbb{S}\ \sigma \\
(StepImp) & \mathtt{a} \Rightarrow_{\mathtt{c}} \mathtt{a}' & \triangleq & \forall \sigma, \mathbb{S}.\, \mathtt{a}\ \mathbb{S}\ \sigma \supset \mathtt{a}'\ \mathtt{Next_c}(\mathbb{S})\ \sigma
\end{array}
$$

Figure 5. Assertion language (abstract from approximation)

$\boxed{\Psi_G \vdash \{\mathtt{a}\}\,\mathbb{P}}$ **(*Well-formed Program*)**

$$\frac{\Psi_G \vdash \mathbb{C} : \Psi_G \qquad \Psi_G \vdash \{\mathtt{a}\}\,\mathbb{I} \qquad (\forall \sigma.\ (\forall (\mathtt{f}, \mathtt{a}') \in \Psi_G.\ \mathsf{cptr}\ \mathtt{f}\ \mathtt{a}'\ \sigma) \supset \mathtt{a}\ \mathbb{S}\ \sigma)}{\Psi_G \vdash \{\mathtt{a}\}\,(\mathbb{C}, \mathbb{S}, \mathbb{I})}\ (\textsc{prog})$$

$\boxed{\Psi_{IN} \vdash \mathbb{C} : \Psi}$ **(*Well-formed Code Heap*)**

$$\frac{\mathcal{OK}\ \mathtt{a}_i \qquad \Psi_{IN} \vdash \{\mathtt{a}_i\}\,\mathbb{I}_i \qquad \forall \mathtt{f}_i}{\Psi_{IN} \vdash \{\mathtt{f}_1 \rightsquigarrow \mathbb{I}_1, \ldots, \mathtt{f}_n \rightsquigarrow \mathbb{I}_n\} : \{\mathtt{f}_1 \rightsquigarrow \mathtt{a}_1, \ldots, \mathtt{f}_n \rightsquigarrow \mathtt{a}_n\}}\ (\textsc{cdhp})$$

$\boxed{\Psi \vdash \{\mathtt{a}\}\,\mathbb{I}}$ **(*Well-formed Instruction Sequence*)**

$$\frac{\begin{array}{l}(\lambda(\mathbb{H}, \mathbb{R}), \sigma.\, \mathbb{R}(\mathtt{r}_s) \le i\ \wedge\ \mathtt{a}\ (\mathbb{H}, \mathbb{R})\ \sigma) \Rightarrow \mathtt{a}' \\ (\lambda(\mathbb{H}, \mathbb{R}), \sigma.\, \mathbb{R}(\mathtt{r}_s) > i\ \wedge\ \mathtt{a}\ (\mathbb{H}, \mathbb{R})\ \sigma) \Rightarrow \Psi(\mathtt{f}) \\ \Psi \vdash \{\mathtt{a}'\}\,\mathbb{I} \qquad \mathtt{f} \in \mathsf{dom}(\Psi)\end{array}}{\Psi \vdash \{\mathtt{a}\}\ \mathsf{bgti}\ \mathtt{r}_s, i, \mathtt{f}; \mathbb{I}}\ (\textsc{bgti})$$

$$\frac{\mathtt{a} \Rightarrow (\lambda(\mathbb{H}, \mathbb{R}), \sigma.\, \exists \mathtt{a}'.\, \mathtt{a}'\ (\mathbb{H}, \mathbb{R})\ \sigma\ \wedge\ \mathsf{cptr}\ \mathbb{R}(\mathtt{r})\ \mathtt{a}'\ \sigma)}{\Psi \vdash \{\mathtt{a}\}\,\mathsf{jmp}\ \mathtt{r}}\ (\textsc{jmp})$$

$$\frac{\begin{array}{l}(\lambda \mathbb{S}, \sigma.\, \mathsf{cptr}\ \mathtt{f}\ \Psi(\mathtt{f})\ \sigma \wedge \mathtt{a}\ \mathbb{S}\ \sigma) \Rightarrow \mathtt{a}' \\ \mathtt{f} \in \mathsf{dom}(\Psi) \qquad \Psi \vdash \{\mathtt{a}'\}\,\mathbb{I}\end{array}}{\Psi \vdash \{\mathtt{a}\}\,\mathbb{I}}\ (\textsc{ecp})$$

Figure 6. Changed inference rules (abstract from approximation)

Since the *cptr* predicate is packed in σ, we define the following "dummy" cptr predicate to automate the task of unpacking σ, pick out and apply *cptr*, etc.,

$$\mathsf{cptr}\ \mathtt{f}\ \mathtt{a}\ \langle Spec, (cptr, \Psi)\rangle \triangleq cptr\ \mathtt{f}\ (\lambda \mathbb{S}, \Psi'.\ \mathtt{a}\ \mathbb{S}\ \langle Spec, (cptr, \Psi')\rangle)\ \Psi$$

and can easily define a relaxed codeptr predicate similar to the one in [16]

$$\mathsf{codeptr}\ \mathtt{f}\ \mathtt{a}\ \sigma \triangleq \exists\, \mathtt{a}'.\, \mathsf{cptr}\ \mathtt{f}\ \mathtt{a}'\ \sigma\ \wedge\ \mathtt{a} \Rightarrow \mathtt{a}'.$$

We present a set of abstract inference rules in Figure 6. Given the similarity between our assertion logic and XCAP's, these rules are very similar, with all the changes highlighted. Rules (SEQ) and (JD) are unchanged and omitted.

In rule (CDHP) we now require the following check on every assertion

$$\mathcal{OK}\ \mathtt{a} \triangleq \forall \sigma, \sigma'.\, (\forall \mathtt{f}, \mathtt{a}'.\, \mathsf{cptr}\ \mathtt{f}\ \mathtt{a}'\ \sigma \supset \mathsf{cptr}\ \mathtt{f}\ \mathtt{a}'\ \sigma') \supset \forall \mathbb{S}.\, \mathtt{a}\ \mathbb{S}\ \sigma \supset \mathtt{a}\ \mathbb{S}\ \sigma'$$

to make sure that it is monotonic over the growth of code heap specification. It is up to the programmer to decide how to achieve $\mathcal{OK}$ for certain assertion a. In general he can turn any a into one that satisfies $\mathcal{OK}$ by the following lifting:

$$\lambda \mathbb{S}.\, \lambda \sigma.\, \forall \sigma'.\, (\forall \mathtt{f}, \mathtt{a}'.\, \mathsf{cptr}\ \mathtt{f}\ \mathtt{a}'\ \sigma \supset \mathsf{cptr}\ \mathtt{f}\ \mathtt{a}'\ \sigma')\ \supset\ \mathtt{a}\ \mathbb{S}\ \sigma'.$$

We discuss the usage of $\mathcal{OK}$ in the next section.

For top-level rule (PROG), since our code heap specification is now abstract, we do not have a concrete code heap specification to pack and supply to assertions. We instead require the initial assertions to hold on all concrete code heap specifications that contain all the code pointers in the abstract specification. In practice, many programs' initial precondition is empty anyway, which makes this trivially true. The derived rules (WEAKEN) and (LINK) in Sect. 2 still hold.

The following are the soundness and partial-correctness theorems. They are proved in the next section by building "semantic" approximation. Since our code heap specification is not specified pointwise, the theorem below only guarantees partial-correctness at control-flow transfer points. This is by no means a limitation of our approach. One can easily alter the machine model and inference rules to allow the code heap specification to be more detailed or point-wise.

THEOREM 3.1 (Soundness).
If $\Psi \vdash \{\mathtt{a}\}\,\mathbb{P}$ then for any number n there exists $\mathbb{P}'$ such that $\mathbb{P} \longmapsto^n \mathbb{P}'$.

THEOREM 3.2 (Partial-Correctness).
If $\Psi \vdash \{\mathtt{a}\}\,\mathbb{P}$ then for any n there exists $(\mathbb{C}, \mathbb{S}, \mathbb{I})$ such that $\mathbb{P} \longmapsto^n (\mathbb{C}, \mathbb{S}, \mathbb{I})$ and

1. if $\mathbb{I} = \mathsf{jd}\ \mathtt{f}$ then there exists σ such that $\Psi(\mathtt{f})\ \mathbb{S}\ \sigma$;
2. if $\mathbb{I} = \mathsf{jmp}\ \mathtt{r}$ then there exists σ such that $\Psi(\mathbb{S}.\mathbb{R}(\mathtt{r}))\ \mathbb{S}\ \sigma$;
3. if $\mathbb{I} = \mathsf{bgti}\ \mathtt{r}_s, i, \mathtt{f}$ and $\mathbb{S}.\mathbb{R}(\mathtt{r}_s) > i$ then there exists σ such that $\Psi(\mathtt{f})\ \mathbb{S}\ \sigma$.

There is *no* restriction on how these theorems should be proved. Other than the approximation-based approach used in the next section, there could be different ways to do so. Indeed, our abstract specification and inference rules are very general and truly independent from the underlying meta-theory proof.

Our abstract language enjoys stronger expressive power than XCAP [16, 18]. All XCAP code can be certified in our language. The reader can refer to [16, 18] for examples of how ECP can be used in logic-based verification.

Figure 7 shows coq code for this section, excluding target machine and auxiliary library. These are all that a programmer would ever need to see and use.

```
Definition X := {Spec : _ & ((Label -> (State -> Spec -> Prop) -> Spec -> Prop) * Spec) %type}.

Definition PropX     := X -> Prop.

Definition Assertion := State -> PropX.

Definition CdHpSpec  := Map Label Assertion.

Notation "a ==> b" := (forall s x, (a : Assertion) s x -> (b : Assertion) s x)
                      (at level 70, right associativity).

Definition cptr f (a : Assertion) x := match x with existT Spec (pair cptr Si) =>
  cptr f (fun S Si => a S (existT _ Spec (pair cptr Si))) Si
end.

Definition codeptr f a x := exists a', cptr f a' x /\ a ==> a'.

Definition ok A (a : A -> PropX) :=
  forall x y, (forall f a, cptr f a x -> cptr f a y) -> forall s, a s x -> a s y.
Implicit Arguments ok [A].

Inductive WFiseq : CdHpSpec -> Assertion -> InstrSeq -> Prop :=
  | wfiseq : forall Si a c I a', a ==> (fun s x => exists s', Next c s s' /\ a' s' x) ->
               WFiseq Si a' I                                              -> WFiseq Si a (iseq c I)
  | wfbgt  : forall Si a rs rt f I (a' a'' : Assertion),
               (forall s x, (_R s rs <= _R s rt /\ a s x) -> a'' s x) ->
               (forall s x, (_R s rs >  _R s rt /\ a s x) -> a'  s x) ->
               lookup Si f a' -> WFiseq Si a'' I                  -> WFiseq Si a (bgt rs rt f I)
  | wfbgti : forall Si a rs w f I (a' a'' : Assertion),
               (forall s x, (_R s rs <= w /\ a s x) -> a'' s x) ->
               (forall s x, (_R s rs >  w /\ a s x) -> a'  s x) ->
               lookup Si f a' -> WFiseq Si a'' I                  -> WFiseq Si a (bgti rs w f I)
  | wfjd   : forall Si a f a', lookup Si f a' -> a ==> a'                  -> WFiseq Si a (jd f)
  | wfjmp  : forall Si a r, a ==> (fun s x => exists a', cptr (_R s r) a' x /\ a' s x)
                                                                          -> WFiseq Si a (jmp r)
  | wfecp  : forall Si a f a' a'' I, WFiseq Si a' I -> lookup Si f a'' ->
               (fun s x => cptr f a'' x /\ a s x) ==> a'                        -> WFiseq Si a I.

Definition WFcode Si C (Si' : CdHpSpec) :=
  forall f a, lookup Si' f a -> ok a /\ exists I, Map.lookup C f I /\ WFiseq Si a I.

Definition WFprog Si a P := match P with pair C (pair s i) =>
  WFcode Si C Si /\ WFiseq Si a i /\ forall x, (forall f a, lookup Si f a -> cptr f a x) -> a s x
end.
```

Figure 7. The complete coq code a programmer would see and use

4. An Approximation Model and Meta-Theory Proofs

In this section we build an approximation-based model for ECPs to establish the soundness and partial-correctness theorems. The full detail can be found in [15].

The basic idea of approximation is, if one wants to prove the sequence

$$\mathtt{a}_1\ \mathbb{S}_1 \supset \mathtt{a}_2\ \mathbb{S}_2 \supset \ldots \supset \mathtt{a}_n\ \mathbb{S}_n \supset \ldots$$

and for some reason can not specify or prove it, one might instead be able to add a natural number index to $\{\mathtt{a}_i\}$ and prove the following sequences

$$\mathtt{a}_1\ m\ \mathbb{S}_1 \supset \mathtt{a}_2\ (m-1)\ \mathbb{S}_2 \supset \ldots \supset \mathtt{a}_n\ (m-n+1)\ \mathbb{S}_n \supset \ldots$$

for every m. If that is the case, then one can easily get the following sequence

$$(\forall m.\ \mathtt{a}_1\ m\ \mathbb{S}_1) \supset (\forall m.\ \mathtt{a}_2\ m\ \mathbb{S}_2) \supset \ldots \supset (\forall m.\ \mathtt{a}_n\ m\ \mathbb{S}_n) \supset \ldots$$

and in many cases $(\forall m.\ \mathtt{a}_i\ m\ \mathbb{S}_i)$ is strong enough or as good as $(\mathtt{a}_i\ \mathbb{S}_i)$.

We define a "concrete" approximation-based specification language as well as translations from the abstract specifications in the last section in Figure 8. These constructs are "concrete" because they all involve certain details of our approximation model. Although this kind of concrete specification constructs are directly used by some other approximation-based work as their real program specifications, here concrete specifications and proofs are only used for the purpose of establishing soundness and partial-correctness guarantees, and should not be revealed to end programmers. As discussed in the previous section, abstract assertion and program logics are all they need in order to carry out verifications.

The collection type of indexed code heap specification, ${}_cSpec$ is similar to $dSpec$ in Section 2. The dependent pair of ${}_cSpec$ and its index forms the baseline concrete code heap specification type

$$
\begin{array}{rrcl}
({}_cSpec\,0) & \phi & \in & \mathsf{unit} \\
({}_cSpec\,(n{+}1)) & \phi & \in & ({}_cSpec\,n) \\
 & & & \times(Label \rightharpoonup \\
 & & & \quad (State \to {}_cSpec\,n \to Prop)) \\
({}_iSpec) & \Phi & \in & \Sigma n.\ {}_cSpec\,n \\
({}_i\mathit{Assertion}) & \mathsf{b} & \in & State \to {}_iSpec \to Prop \\
({}_iCdHpSpec) & \psi & ::= & \{\mathtt{f} \rightsquigarrow \mathsf{b}\}^* \\
(AssertTr) & [\![\mathsf{a}]\!] & \triangleq & \lambda\mathbb{S}, \Phi.\ \mathsf{a}\ \mathbb{S}\ \langle {}_iSpec, ({}_i\mathsf{cptr}, \Phi)\rangle \\
(CdSpecTr) & [\![\{\mathtt{f} \rightsquigarrow \mathsf{a}\}^*]\!] & \triangleq & \{\mathtt{f} \rightsquigarrow [\![\mathsf{a}]\!]\}^*
\end{array}
$$

Figure 8. Concrete approximation-based specification language

${}_iSpec$. With the concrete assertion type being $State \to {}_iSpec \to Prop$. A more general concrete code heap specification is ${}_iCdHpSpec$. But the assertions in it can not directly take itself as an argument. Instead, we define the following "cap" function to construct assertion-friendly ${}_iSpec$ specifications by only keeping a finite number of indicies in it.

$$\lfloor\psi\rfloor_0 \triangleq \mathsf{tt}$$

$$\lfloor\psi\rfloor_{n+1} \triangleq (\lfloor\psi\rfloor_n, \{\mathtt{f} \mapsto \lambda\mathbb{S}.\, \lambda\phi.\ \psi(\mathtt{f})\ \mathbb{S}\ \langle n, \phi\rangle\})$$

We define the concrete monotonicity condition as follows:

$${}_i\mathcal{OK}\ \mathsf{b} \triangleq \forall\mathbb{S}, \psi, n.\ \mathsf{b}\ \mathbb{S}\ \langle n+1, \lfloor\psi\rfloor_{n+1}\rangle \supset \mathsf{b}\ \mathbb{S}\ \langle n, \lfloor\psi\rfloor_n\rangle$$

and prove the following lemma to go from abstract to concrete monotonicity.

LEMMA 4.1 ($\mathcal{OK}$ to ${}_i\mathcal{OK}$ Preservation).
If $\mathcal{OK}\ \mathsf{a}$ then ${}_i\mathcal{OK}\ [\![\mathsf{a}]\!]$.

The concrete embedded code pointers predicate ${}_i\mathsf{cptr}$ is then defined as

$${}_i\mathsf{cptr}\ \mathtt{f}\ \mathsf{b}\ (\langle n, \phi\rangle : {}_iSpec) \triangleq {}_i\mathcal{OK}\ \mathsf{b} \wedge \exists\psi.\ \psi(\mathtt{f}) = \mathsf{b} \wedge \lfloor\psi\rfloor_n = \phi$$

The abstract top-level program well-formedness rule is transformed into a concrete rule with the change on the well-formedness of state.

$$\frac{\Psi_G \vdash \mathbb{C} : \Psi_G \qquad \Psi_G \vdash \{\mathsf{a}\}\,\mathbb{I} \qquad (\forall n.\ [\![\mathsf{a}]\!]\ \mathbb{S}\ \lfloor[\![\Psi_G]\!]\rfloor_n)}{\Psi_G \vdash \{\mathsf{a}\}\,(\mathbb{C}, \mathbb{S}, \mathbb{I})}\ (\text{PROG'})$$

The following lemma shows that the abstract rule entails the concrete rule.

LEMMA 4.2 (Rule (PROG) to Rule (PROG') Preservation).
If $\Psi \vdash_{PROG} \{\mathsf{a}\}\ \mathbb{P}$ then $\Psi \vdash_{PROG'} \{\mathsf{a}\}\ \mathbb{P}$.

With the concrete approximation model defined in this section, we are able to show that the above invariant gets preserved during the execution, in particular,

$$(\forall n.\ [\![\mathsf{a}_1]\!]\ \mathbb{S}_1\ \lfloor[\![\Psi_G]\!]\rfloor_n) \supset (\forall n.\ [\![\mathsf{a}_2]\!]\ \mathbb{S}_2\ \lfloor[\![\Psi_G]\!]\rfloor_n) \supset \ldots$$

and finally prove the soundness and partial-correctness theorems.

Further comparison with XCAP. By avoiding an extra layer of syntax, we can directly reuse many of Coq's built-in features. For example, for program specification we automatically have full support of **impredicative polymophisms** (semi-supported in XCAP, elimination disallowed) and **inductive definitions** (not supported for $PropX$ in XCAP). Many of Coq's **built-in proof tactics** are much more effective now than on the syntactic $PropX$ constructs in XCAP.

The new approach presented in this paper effectively reduces the implementation overhead XCAP has encountered. For example, the old $PropX$ meta theory, which is about 5,000 lines of Coq code [16, 18], is replaced by less than 300 lines of code. More importantly, unlike the old implementation, the new approach **no longer** needs to use de Bruijn indices to represent impredicative polymorphisms. Overall, we expect the simplification in program specification and proof will reduce the lines of those code in [18] by 60% to 80%.

Similar to previous work [16, 3], we assume the support of impredicative polymorphisms and dependent types in the (mechanized) meta logic.

Monotonicity is the only additional cost, comparing to the syntactic approach such as XCAP. It is easy to prove when the usage of implication in program specifications is limited. As the next section will show, this monotonicity will be used anyway to support dynamic code heap generation.

5. Supporting Dynamic Code Generation

XCAP, as well as the discussion presented in the previous sections, only talks about constant code heap programs. Given the approximation-based treatment of embedded code pointers in our new framework, we are able to specify and reason about meta-properties, such as $\mathcal{OK}$ (monotonicity over growth of code heap), of specifications, and easily support dynamic code generation. In this section we present one simple example of this. We expect that the raw power of our approximation model should suffice for other more complex and useful dynamic code generation and updating, as well as self-modifying code in general.

The target machine in Sect. 2 does not support dynamic code generation. We extend it with the following virtual dynamic code generation instruction.

if $\mathbb{I} =$	then $(\mathbb{C}, (\mathbb{H}, \mathbb{R}), \mathbb{I}) \longmapsto$
$\mathsf{loadcode}\ \mathtt{r}_t[\mathbb{I}']; \mathbb{I}''$	$(\mathbb{C}\{\mathtt{f} \mapsto \mathbb{I}'\}, (\mathbb{H}, \mathbb{R}\{\mathtt{r}_t \rightsquigarrow \mathtt{f}\}), \mathbb{I}'')$ where $\mathtt{f} \notin \mathsf{dom}(\mathbb{C})$

Apparently, loadcode does not correspond to any single physical instruction, as it automatically finds an available code address and writes an instruction sequence $\mathbb{I}'$ into the code heap in one step. Note, however, that this does not make the support of dynamic code generation any easier.

To certify this single instruction, we add the following new abstract instruction sequence rule to join those presented in Figure 6.

$$\frac{\begin{array}{l}\mathcal{OK}\ \mathsf{a}' \qquad \Psi \vdash \{\mathsf{a}'\}\,\mathbb{I} \qquad \Psi \vdash \{\mathsf{a}''\}\,\mathbb{I}' \\ (\lambda(\mathbb{H}, \mathbb{R}), \sigma.\ \mathsf{cptr}\ \mathbb{R}(\mathtt{r}_t)\ \mathsf{a}'\ \sigma \wedge \\ \qquad\qquad \exists\mathtt{w}.\ \mathsf{a}\ (\mathbb{H}, \mathbb{R}\{\mathtt{r}_t \rightsquigarrow \mathtt{w}\})\ \sigma)\ \Rightarrow\ \mathsf{a}''\end{array}}{\Psi \vdash \{\mathsf{a}\}\ \mathsf{loadcode}\ \mathtt{r}_t[\mathbb{I}]; \mathbb{I}'}\ (\text{LOADCODE})$$

To load a code block into the code heap, one has to provide a valid precondition under which the code block is well-formed. This is the only change to the abstract assertion and program logics.

The approximation model we build in the previous section is already powerful enough for dynamic code generations. We only need to change the concrete top-level invariant rule to the following

$$\frac{\Psi_{_G} \subseteq \Psi \qquad \Psi \vdash \mathbb{C} : \Psi \qquad \Psi \vdash \{\mathtt{a}\}\,\mathbb{I} \qquad (\forall \Psi', n.\ (\Psi \subseteq \Psi') \supset [\![\mathtt{a}]\!]\ \mathbb{S}\ \lfloor [\![\Psi']\!] \rfloor_n)}{\Psi_{_G} \vdash \{\mathtt{a}\}\,(\mathbb{C}, \mathbb{S}, \mathbb{I})}\ (\text{PROG"})$$

The soundness and partial-correctness theorems still hold, except that the latter needs some small changes as follows.

THEOREM 5.1 (Partial-Correctness (Dynamic Code Generation)).

If $\Psi \vdash \{\mathtt{a}\}\,\mathbb{P}$ then for any n there exists $(\mathbb{C}, \mathbb{S}, \mathbb{I})$ such that $\mathbb{P} \longmapsto^n (\mathbb{C}, \mathbb{S}, \mathbb{I})$ and

1. if $\mathbb{I} = \mathsf{jd}\,\mathtt{f}$ and $\mathtt{f} \in \mathsf{dom}(\Psi)$ then there exists σ such that $\Psi(\mathtt{f})\ \mathbb{S}\ \sigma$;
2. if $\mathbb{I} = \mathsf{jmp}\,\mathtt{r}$ and $\mathbb{S}.\mathbb{R}(\mathtt{r}) \in \mathsf{dom}(\Psi)$then there exists σ such that $\Psi(\mathbb{S}.\mathbb{R}(\mathtt{r}))\ \mathbb{S}\ \sigma$;
3. if $\mathbb{I} = \mathsf{bgti}\,\mathtt{r}_s, i, \mathtt{f}, \mathbb{S}.\mathbb{R}(\mathtt{r}_s) > i$, and $\mathtt{f} \in \mathsf{dom}(\Psi)$ then there exists σ such that $\Psi(\mathtt{f})\ \mathbb{S}\ \sigma$.

6. Related Work and Conclusion

GTAL [8] is a logic-based assembly verification system. Its assertion logic is a mixture of higher-order logic, dependent types, linear types, embedded code pointers, as well as recursive types. GTAL does not support general references. Unlike XCAP, GTAL does not lift meta logic propositions into its assertion logic. Thus it has a much more complex assertion logic. GTAL has been used to certify garbage collector code, while the mutator code is certified by a smaller type language defined in GTAL. Its meta theory has not been mechanized.

Spec# [4] extends C# language's type system with more expressive constructs, including (machine-)logical pre- and post-conditions, verified either statically by automatic theorem prover or dynamically by run-time checking. The potential use of machine-logical specification and dynamic checking makes Spec# a bit different from other approaches mentioned in this paper.

Hoare type theory [13, 12] is a stateful dependent type system with embedded logical assertions. Similar to XCAP, it lifts logical assertions and proofs into the type systems. However, HTT starts with a functional language instead of assembly languages. HTT supports embedded code pointers that can be used to certify high-level effectful programs. HTT currently does not support general references and recursive types. Its meta theory has not been mechanized.

Cai et al. [5] shares one common goal with us: to build an approximation-based model for embedded code pointers with partial-correctness. After building their model, they also adapt the XCAP set of inference rules. The model they built does not rely on dependent types, which is an advantage over our model. However, we expect that their model will not support dynamic growth of the code heap (and the data heap, when similar modeling is built for general reference) with the presence of impredicative polymorphisms. Their approximation details, although slightly simpler, is explicit in program specifications and proofs. They also support recursive types with the same model.

Future work. Our approach should be applicable to other higher-order features such as general references and recursive types. It will also be interesting to see how these various features should be optimally mixed in a same system, so that the abstract-concrete separation be best preserved.

Given the simplicity of our solution, it should be much easier to certify code that is similar to those in [18]. It is our goal to combine the new solution for embedded code pointers with other improvements on interactive theorem proving to certify more systems code, with at least one magnitude higher productivity.

Conclusion. We present a new approach to support embedded code pointers with partial-correctness in logic-based verification. Our approach utilizes approximation technology to establish the soundness and partial-correctness guarantee. All approximation details are abstracted out and hidden from the programmer. Our work reaches a nice balance between existing syntactic and approximation-based approaches and will also improve the productivity of logic-based verification.

References

[1] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. 27th ACM Symposium on Principles of Programming Languages*, pages 243–253, Jan. 2000.

[2] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, Sept. 2001.

[3] A. W. Appel, P.-A. Mellies, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *Proc. 34th ACM Symposium on Principles of Programming Languages*, pages 109–122, Nice, France, Jan. 2007.

[4] M. Barnett, K. R. M. Leino, , and W. Schulte. The spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*. Springer-Verlag, 2004.

[5] H. Cai, X. Feng, Z. Shao, and G. Tan. Towards logical reasoning about code pointers. Unpublished manuscript; Tsinghua University.

[6] J. Chen, C. Hawblitzel, F. Perry, M. Emmi, J. Condit, D. Coetzee, and P. Pratikakis. Type-preserving compilation for large-scale optimizing object-oriented compilers. In *Proc. 2008 ACM Conference on Programming Language Design and Implementation*, page to appear. ACM Press, 2008.

[7] R. W. Floyd. Assigning meaning to programs. *Communications of the ACM*, Oct. 1967.

[8] C. Hawbilitzel, H. Huang, L. Wittie, and J. Chen. A garbage-collecting typed assembly language. In *Proc. 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, New York, NY, USA, Jan. 2007. ACM Press.

[9] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, Oct. 1969.

[10] A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. In *Proc. 2007 ACM Conference on Programming Language Design and Implementation*, pages 468–479, San Diego, CA, June 2007.

[11] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, pages 85–97. ACM Press, Jan. 1998.

[12] A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable adts in hoare type theory. In *Proc. 2007 European Symposium on Programming*, pages 189–204, Braga, Portugal, Mar. 2007.

[13] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in hoare type theory. In *Proc. 11th ACM SIGPLAN International Conference on Functional Programming*, pages 62–73, Portland, OR, USA, Sept. 2006. ACM Press.

[14] G. Necula. Proof-carrying code. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 106–119, New York, Jan. 1997. ACM Press.

[15] Z. Ni. Implementation for an abstract, approximation-based approach to embedded code pointers and partial-correctness. `http://research.microsoft.com/~nzz/`, Feb. 2008.

[16] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. 33rd ACM Symposium on Principles of Programming Languages*, pages 320–333, Charleston, South Carolina, Jan. 2006.

[17] Z. Ni and Z. Shao. A translation from typed assembly languages to certified assembly programming. Technical Report `http://flint.cs.yale.edu/flint/publications/talcap.html`, Dept. of Computer Science, Yale Univ., New Haven, CT, Nov. 2006.

[18] Z. Ni, D. Yu, and Z. Shao. Using XCAP for systems programming: Machine context management. In *Proc. 20th International Conference on Theorem Proving in Higher Order Logics*, pages 189–206, Kaiserslautern, Germany, Sept. 2007.