A Translation from Typed Assembly Languages to Certified Assembly Programming

Zhaozhong Ni[1] Zhong Shao[2]

[1] Microsoft Research, One Microsoft Way
Redmond, WA 98052, U.S.A.
zhaozhong.ni@microsoft.com
[2] Department of Computer Science,Yale University
New Haven, CT 06520-8285, U.S.A.
shao@cs.yale.edu

Abstract. Typed assembly languages (TAL) and certified assembly programming (CAP) are two techniques for low-level verifications. TAL uses syntactic types and typing rules for program specification and reasoning, while CAP uses Hoare-style logic assertions and semantic subsumptions. They are suitable for different kinds of verification tasks. Previously, programs verified in either one of them can not interoperate freely with the other, making it hard to integrate them into a complete system. Moreover, the relationship between TAL and CAP lines of work has not been discussed extensively.
In this paper, we do so by presenting a translation from a TAL language to a CAP language. The translation involves an intermediate step of a "semantic" TAL language. To better illustrate our key points, all three languages share a same untyped machine. Thus the shape of the source language is slightly different from the original TAL's. Nevertheless, it supports polymorphic code, mutable reference, existential, and recursive types. The target language is an extension of XCAP, a recent CAP language, with support of logical recursive specifications and logical mutable reference, both require minimal changes to XCAP and its meta theory. Since we proved typing preservation for the translation, there is a clear path to link and interoperate well-typed programs from traditional certifying compilers with certified libraries by CAP-like systems.

1 Introduction

Proof-carrying code (PCC) [1] is a general framework that can, in principle, verify safety properties of arbitrary machine-language programs. Existing PCC systems [2–5], however, have focused on programs written in type-safe languages [6, 7] or variants of typed assembly languages (TAL) [8]. Type-based approaches are attractive because they facilitate automatic generation of the safety proofs (by using certifying compilers) and provide great support to modularity and higher-order language features. But they also suffer from several serious limitations. First, types are not expressive enough to specify sophisticated invariants commonly seen in the verification of low-level system software. Recent work on *logic-based type systems* [9–13] have made types more expressive but they still cannot specify advanced state invariants and assertions [14–16] definable in

a general-purpose predicate logic with inductive definitions [17]. Second, type systems are too weak to prove advanced properties and program correctness, especially in the context of concurrent assembly code [15, 18]. Finally, different style languages often require different type systems, making it hard to reason about interoperability.

An alternative to type-based methods is to use Hoare logic [19, 20]—a widely applied technique in program verification. Hoare logic supports formal reasoning using very expressive assertions and inference rules from a general-purpose logic. In the context of foundational proof-carrying code (FPCC) [21, 22], the assertion language is often unified with the mechanized meta logic (following Gordon [23])—proofs for Hoare logic consequence relation and Hoare-style program derivations are explicitly written out in a proof assistant. For example, Touchstone PCC [24] used Hoare-style assertions to express complex program invariants. Appel *et al* [25, 26] used Hoare-style state predicates to construct a general semantic model for machine-level programs. Shao *et al* [22, 16, 18, 27] recently developed a certified assembly programming (CAP) framework that uses Hoare-style reasoning to verify low-level system libraries and general multi-threaded assembly programs.

TAL and CAP are suitable for different kinds of verification tasks. Previously, programs verified in either one of them can not interoperate freely with the other, making it hard to integrate them into a complete system. Moreover, the relationship between TAL and CAP lines of work has not been discussed extensively.

In this paper we compare the type-based and logic-based methods by presenting a type-preserving translation from a TAL language to a CAP langyage. The translation involves an intermediate step of a "semantic" TAL language. Our translation supports polymorphic code, mutable reference, existential, and recursive types. The target language is an extension of XCAP [27], a recent CAP language which supports embedded code pointers and impredicative polymorphisms. We added to XCAP the support of logical recursive specifications and logical mutable reference, both require minimal changes to XCAP and its meta theory, and call our new language XCAP+. Since we proved typing preservation for the translation from TAL to XCAP+, there is a clear path to link and interoperate well-typed programs from traditional certifying compilers with certified libraries by CAP-like systems.

We present the paper in the order of source language TAL (Sec 2), intermediate language STAL (Sec 3), and target language XCAP+ (Sec 4). Then we present the complete translation and the type-preservation properties (Sec 5). Finally we do some brief discussion and conclude (Sec 6).

2 Source Language: TAL

In this section we present a typed assembly language (TAL). To make the comparison easier, we build all languages in this paper on a common target machine. Throughout the paper, we also assume an underlying mechanized meta logic like

$$(\mathit{Prop})\ p,q ::= \mathsf{True} \mid \mathsf{False} \mid \neg p \mid p \wedge q \mid p \vee q \mid p \supset q \mid \forall x{:}A.\, p \mid \exists x{:}A.\, p \mid \ldots$$

$$
\begin{array}{rrl}
(\textit{Program}) & \mathbb{P} & ::= (\mathbb{C},\mathbb{S},\mathbb{I}) \\
(\textit{State}) & \mathbb{S} & ::= (\mathbb{H},\mathbb{R}) \\
(\textit{Heap}) & \mathbb{H} & ::= \{\mathtt{l} \rightsquigarrow \mathtt{w}\}^* \\
(\textit{RegFile}) & \mathbb{R} & ::= \{\mathtt{r} \rightsquigarrow \mathtt{w}\}^* \\
(\textit{Register}) & \mathtt{r} & ::= \{\mathtt{rk}\}^{\mathtt{k}\in\{0\ldots31\}} \\
(\textit{Word},\textit{Labels}) & \mathtt{w},\mathtt{f},\mathtt{l} & ::= i \ \ (\textit{nat nums})
\end{array}
\qquad
\begin{array}{rrl}
(\textit{CodeHeap}) & \mathbb{C} & ::= \{\mathtt{f} \rightsquigarrow \mathbb{I}\}^* \\
(\textit{InstrSeq}) & \mathbb{I} & ::= \mathtt{c};\mathbb{I} \mid \mathsf{jd}\ \mathtt{f} \mid \mathsf{jmp}\ \mathtt{r} \\
(\textit{Command}) & \mathtt{c} & ::= \mathsf{bgti}\ \mathtt{r}_s,i,\mathtt{f} \mid \mathsf{addi}\ \mathtt{r}_d,\mathtt{r}_s,i \\
 & & \mid \mathsf{add}\ \mathtt{r}_d,\mathtt{r}_s,\mathtt{r}_t \\
 & & \mid \mathsf{movi}\ \mathtt{r}_d,i \mid \mathsf{mov}\ \mathtt{r}_d,\mathtt{r}_s \\
 & & \mid \mathsf{ld}\ \mathtt{r}_d,\mathtt{r}_s(i) \mid \mathsf{st}\ \mathtt{r}_d(i),\mathtt{r}_s
\end{array}
$$

Fig. 1. Syntax of target machine TM

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 $\mathbb{R}(\mathtt{r}_s) > i$
$\mathtt{c};\mathbb{I}'$	$(\mathbb{C},\mathtt{Next}_{\mathtt{c}}(\mathbb{H},\mathbb{R}),\mathbb{I}')$

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

Fig. 2. Operational semantics of TM

2.1 Target Machine (TM)

We define the syntax and the operational semantics of the target abstract machine (TM) in Fig 1. A complete TM 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 extensions are straightforward. To control the complexity of the paper, we ignore the allocation of memory. The register file is made up of 32 registers and the data heap can potentially be infinite. The operational semantics of this language (see Fig 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."

2.2 Typed Assembly Language (TAL)

The TAL language presented here follows the principle of the original typed assembly languages [8]. However, due to the usage of an untyped raw machine, the shape of typing judgments and rules are slightly different.

Fig 3 presents the type definitions in TAL. Machine word is classified as of value type (τ) including integer, code, tuple, existential package, and recursive data structures. A code heap specification (Ψ) is a partial environment that maps a code label to a "precondition" type ($[\Delta].\Gamma$) for its corresponding code block. Here Δ is a type variable environment and Γ is a register file type which specifies the type for each register.

$\boxed{\Psi \vdash \{[\Delta].\Gamma\}\,\mathbb{P}}$ (***Well-formed Program***)

$$\frac{\vdash \mathbb{C}:\Psi \qquad \Psi \vdash \mathbb{S}:[\Delta].\Gamma \qquad \Psi \vdash \{[\Delta].\Gamma\}\,\mathbb{I}}{\Psi \vdash \{[\Delta].\Gamma\}\,(\mathbb{C},\mathbb{S},\mathbb{I})}\ (\textsc{prog})$$

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

$$\frac{\Psi \vdash \{\Psi(\mathtt{f})\}\,\mathbb{C}(\mathtt{f}) \qquad \forall \mathtt{f} \in \mathsf{dom}(\Psi)}{\vdash \mathbb{C}:\Psi}\ (\textsc{cdhp})$$

$$\begin{array}{rl}
(\textit{CdHpSpec}) & \Psi ::= \{\mathtt{f} \rightsquigarrow [\Delta].\Gamma\}^* \\
(\textit{RfileTy}) & \Gamma ::= \{\mathtt{r} \rightsquigarrow \tau\}^* \\
(\textit{TyVarEnv}) & \Delta ::= \cdot \mid \alpha,\Delta \\
(\textit{WordTy}) & \tau ::= \alpha \mid \mathsf{int} \mid \mathsf{code}[\Delta].\Gamma \\
 & \quad \mid \langle \tau_1,\ldots,\tau_n \rangle \mid \exists\alpha.\,\tau \\
 & \quad \mid \mu\alpha.\,\tau \\
(\textit{DtHpSpec}) & \Phi ::= \{\mathtt{l} \rightsquigarrow \tau\}^*
\end{array}$$

$\boxed{\Psi \vdash \{[\Delta].\Gamma\}\,\mathbb{I}}$ (***Well-formed Instruction Sequence***)

$$\frac{\vdash \{\Gamma\}\,\mathsf{c}\,\{\Gamma'\} \qquad \Psi \vdash \{[\Delta].\Gamma'\}\,\mathbb{I} \qquad \mathsf{c} \in \{\mathsf{add},\mathsf{mov},\mathsf{movi},\mathsf{ld},\mathsf{st}\}}{\Psi \vdash \{[\Delta].\Gamma\}\,\mathsf{c};\mathbb{I}}\ (\textsc{seq})$$

$$\frac{\mathtt{f} \in \mathsf{dom}(\Psi) \qquad \Psi \vdash \{[\Delta].\Gamma\{\mathtt{r}_d \rightsquigarrow \mathsf{code}\ \Psi(\mathtt{f})\}\}\,\mathbb{I}}{\Psi \vdash \{[\Delta].\Gamma\}\,\mathsf{movi}\ \mathtt{r}_d,\mathtt{f};\mathbb{I}}\ (\textsc{movf})$$

$$\frac{\mathtt{f} \in \mathsf{dom}(\Psi) \qquad \vdash [\Delta].\Gamma \leq \Psi(\mathtt{f})}{\Psi \vdash \{[\Delta].\Gamma\}\,\mathsf{jd}\ \mathtt{f}}\ (\textsc{jd}) \qquad \frac{\Gamma(\mathtt{r}) = \mathsf{code}[\Delta'].\Gamma' \qquad \vdash [\Delta].\Gamma \leq [\Delta'].\Gamma'}{\Psi \vdash \{[\Delta].\Gamma\}\,\mathsf{jmp}\ \mathtt{r}}\ (\textsc{jmp})$$

$$\frac{\mathtt{f} \in \mathsf{dom}(\Psi) \qquad \Gamma(\mathtt{r}_s) = \mathsf{int} \qquad \Psi \vdash \{[\Delta].\Gamma\}\,\mathbb{I} \qquad \vdash [\Delta].\Gamma \leq \Psi(\mathtt{f})}{\Psi \vdash \{[\Delta].\Gamma\}\,\mathsf{bgti}\ \mathtt{r}_s,i,\mathtt{f};\mathbb{I}}\ (\textsc{bgti}) \qquad \frac{\vdash [\Delta].\Gamma \leq [\Delta'].\Gamma' \qquad \Psi \vdash \{[\Delta'].\Gamma'\}\,\mathbb{I}}{\Psi \vdash \{[\Delta].\Gamma\}\,\mathbb{I}}\ (\textsc{weaken})$$

Fig. 3. Type definitions and top-level static semantics of TAL

Similarily, a data heap specification (Φ) is a partial environment that maps from a data label to a value type for its corresponding heap cell.

The top-level semantic rules of TAL are also presented in Fig 3. A program is well-formed 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 the precondition $[\Delta].\Gamma$, then executing $\mathbb{I}$ is safe with respect to Ψ. Weakening is allowed to turn one precondition into another provided that they satisfy the subtyping relation. A instruction sequence $\mathtt{c};\mathbb{I}$ is safe if one can find another register file type which serves as both the post-condition of $\mathtt{c}$ and the precondition of $\mathbb{I}$. A direct jump is safe if the current precondition implies the precondition of the target code block specified in Ψ. An indirect jump is safe when the target register is of a code type with a weaker precondition. Constant code labels can be moved into registers.

The remaining typing rules for TAL is presented in Fig 4. Valid subtypings include reducing the registers, instantiation of code type, packing, unpacking, folding, and unfolding. For simple instructions, their pre- and post-conditions do not involve change of type variable environment, thus we only specify the register file types before and after their execution. The well-formed state rule instantiates the current type variable environment and requires a current data heap specification to be supplied and checked. Other judgments and rules are standard and straight-forward.

$\boxed{\vdash [\Delta].\Gamma \leq [\Delta'].\Gamma'}$ ***(Subtyping)***

$$\frac{\Delta \supseteq \Delta' \quad \forall\ \mathtt{r} \in \mathsf{dom}(\Gamma') \quad \Gamma(\mathtt{r}) = \Gamma'(\mathtt{r}) \quad \Delta' \vdash \Gamma'(\mathtt{r})}{\vdash [\Delta].\Gamma \leq [\Delta'].\Gamma'}\ (\textsc{subt}) \qquad \frac{\Gamma(\mathtt{r}) = \mathsf{code}[\alpha, \Delta'].\Gamma' \quad \Delta \vdash \tau'}{\vdash [\Delta].\Gamma \leq [\Delta].\Gamma\{\mathtt{r} : \mathsf{code}[\Delta'].\Gamma'[\tau'/\alpha]\}}\ (\textsc{tapp})$$

$$\frac{\Gamma(\mathtt{r}) = \tau[\tau'/\alpha] \quad \Delta \vdash \tau'}{\vdash [\Delta].\Gamma \leq [\Delta].\Gamma\{\mathtt{r} : \exists\alpha.\,\tau\}}\ (\textsc{pack}) \qquad \frac{\Gamma(\mathtt{r}) = \exists\alpha.\,\tau}{\vdash [\Delta].\Gamma \leq [\alpha, \Delta].\Gamma\{\mathtt{r} : \tau\}}\ (\textsc{unpack})$$

$$\frac{\Gamma(\mathtt{r}) = \tau[\mu\alpha.\,\tau/\alpha]}{\vdash [\Delta].\Gamma \leq [\Delta].\Gamma\{\mathtt{r} : \mu\alpha.\,\tau\}}\ (\textsc{fold}) \qquad \frac{\Gamma(\mathtt{r}) = \mu\alpha.\,\tau}{\vdash [\Delta].\Gamma \leq [\Delta].\Gamma\{\mathtt{r} : \tau[\mu\alpha.\,\tau/\alpha]\}}\ (\textsc{unfold})$$

$\boxed{\vdash \{\Gamma\}\,\mathtt{c}\,\{\Gamma'\}}$ ***(Well-formed Instruction)***

$$\frac{\Gamma(\mathtt{r}_s) = \mathsf{int}}{\vdash \{\Gamma\}\,\mathsf{addi}\ \mathtt{r}_d, \mathtt{r}_s, i\,\{\Gamma\{\mathtt{r}_d : \mathsf{int}\}\}}\ (\textsc{addi}) \qquad \frac{\Gamma(\mathtt{r}_s) = \Gamma(\mathtt{r}_t) = \mathsf{int}}{\vdash \{\Gamma\}\,\mathsf{add}\ \mathtt{r}_d, \mathtt{r}_s, \mathtt{r}_t\,\{\Gamma\{\mathtt{r}_d : \mathsf{int}\}\}}\ (\textsc{add})$$

$$\frac{}{\vdash \{\Gamma\}\,\mathsf{movi}\ \mathtt{r}_d, \mathtt{w}\,\{\Gamma\{\mathtt{r}_d : \mathsf{int}\}\}}\ (\textsc{movi}) \qquad \frac{\Gamma(\mathtt{r}_s) = \tau}{\vdash \{\Gamma\}\,\mathsf{mov}\ \mathtt{r}_d, \mathtt{r}_s\,\{\Gamma\{\mathtt{r}_d : \tau\}\}}\ (\textsc{mov})$$

$$\frac{\Gamma(\mathtt{r}_s) = \langle \tau_1, \ldots, \tau_{\mathtt{w}+1}, \ldots, \tau_n \rangle}{\vdash \{\Gamma\}\,\mathsf{ld}\ \mathtt{r}_d, \mathtt{r}_s(\mathtt{w})\,\{\Gamma\{\mathtt{r}_d : \tau_{\mathtt{w}+1}\}\}}\ (\textsc{ld}) \qquad \frac{\Gamma(\mathtt{r}_d) = \langle \tau_1, \ldots, \tau_{\mathtt{w}+1}, \ldots, \tau_n \rangle \quad \Gamma(\mathtt{r}_s) = \tau_{\mathtt{w}+1}}{\vdash \{\Gamma\}\,\mathsf{st}\ \mathtt{r}_d(\mathtt{w}), \mathtt{r}_s\,\{\Gamma\}}\ (\textsc{st})$$

$\boxed{\Delta \vdash \tau \quad \Psi \vdash \mathbb{S} : [\Delta].\Gamma \quad \Psi \vdash \mathbb{H} : \Phi \quad \Psi; \Phi \vdash \mathbb{R} : \Gamma \quad \Psi; \Phi \vdash \mathtt{w} : \tau}$

(Well-formed Type, State, Heap, Register file, and Word)

$$\frac{FTV(\tau) \subseteq \Delta}{\Delta \vdash \tau}\ (\textsc{type}) \qquad \frac{\cdot \vdash \tau_i \quad \forall\, i \quad \Psi \vdash \mathbb{H} : \Phi \quad \Psi; \Phi \vdash \mathbb{R} : \Gamma[\tau_1, \ldots, \tau_n/\alpha_1, \ldots, \alpha_n]}{\Psi \vdash (\mathbb{H}, \mathbb{R}) : [\alpha_1, \ldots, \alpha_n].\Gamma}\ (\textsc{state})$$

$$\frac{\Psi; \Phi \vdash \mathbb{H}(\mathtt{l}) : \Phi(\mathtt{l}) \quad \forall\, \mathtt{l} \in \mathsf{dom}(\Phi)}{\Psi \vdash \mathbb{H} : \Phi}\ (\textsc{heap}) \qquad \frac{\Psi; \Phi \vdash \mathbb{R}(\mathtt{r}) : \Gamma(\mathtt{r}) \quad \forall\, \mathtt{r} \in \mathsf{dom}(\Gamma)}{\Psi; \Phi \vdash \mathbb{R} : \Gamma}\ (\textsc{rfile})$$

$$\frac{}{\Psi; \Phi \vdash \mathtt{w} : \mathsf{int}}\ (\textsc{int}) \qquad \frac{\mathtt{f} \in \mathsf{dom}(\Psi)}{\Psi; \Phi \vdash \mathtt{f} : \mathsf{code}\ \Psi(\mathtt{f})}\ (\textsc{code}) \qquad \frac{\cdot \vdash \tau' \quad \Psi; \Phi \vdash \mathtt{f} : \mathsf{code}[\alpha, \Delta].\Gamma}{\Psi; \Phi \vdash \mathtt{f} : \mathsf{code}[\Delta].\Gamma[\tau'/\alpha]}\ (\textsc{poly})$$

$$\frac{\Phi(\mathtt{l}+i-1) = \tau_i \quad \forall\, i}{\Psi; \Phi \vdash \mathtt{l} : \langle \tau_1, \ldots, \tau_n \rangle}\ (\textsc{tup}) \qquad \frac{\cdot \vdash \tau' \quad \Psi; \Phi \vdash \mathtt{w} : \tau[\tau'/\alpha]}{\Psi; \Phi \vdash \mathtt{w} : \exists\alpha.\,\tau}\ (\textsc{ext}) \qquad \frac{\Psi; \Phi \vdash \mathtt{w} : \tau[\mu\alpha.\,\tau/\alpha]}{\Psi; \Phi \vdash \mathtt{w} : \mu\alpha.\,\tau}\ (\textsc{rec})$$

Fig. 4. Static semantics of TAL (continued)

The soundness theorem guarantees that given a well-formed program, the machine will never get stuck. It is proved following the syntactic approach of proving type soundness [28]. We also list a few key lemmas below.

Theorem 1 (TAL Soundness). If $\Psi \vdash \{[\Delta].\Gamma\}\,\mathbb{P}$, then for all natural number n, there exists a program $\mathbb{P}'$ such that $\mathbb{P} \longmapsto^n \mathbb{P}'$.

Lemma 1 (State Weakening).
If $\Psi \vdash \mathbb{S} : [\Delta].\Gamma$ and $\vdash [\Delta].\Gamma \leq [\Delta'].\Gamma'$ then $\Psi \vdash \mathbb{S} : [\Delta'].\Gamma'$.

Lemma 2 (Instruction Typing).
If $\Psi \vdash \mathbb{S} : [\Delta].\Gamma$ and $\vdash \{\Gamma\}\mathtt{c}\{\Gamma'\}$ then $\Psi \vdash \mathtt{Next_c}(\mathbb{S}) : [\Delta].\Gamma'$.

3 A "Semantic" TAL Language

Instead of doing translation from TAL to CAP directly, we create a new "Semantic" TAL language (STAL) to serve as an intermediate step between them. Let us revisit the static semantics of TAL in Figure 4 and the lemmas used in TAL soundness proof.

First look at the set of subtyping rules between preconditions ($\vdash [\Delta].\Gamma \leq [\Delta'].\Gamma'$). If we also look at the State Weakening lemma (Lemma 1) in TAL soundness proof, it is easy to see that all of those syntactic rules are just used for the meta implication between the two state typings in the soundness proof. Given a mechanized meta logic, we can replace these rules with a single one,

$$\frac{\Psi \vdash \mathbb{S} : [\Delta].\Gamma \ \supset \ \Psi \vdash \mathbb{S} : [\Delta'].\Gamma' \qquad \forall\, \mathbb{S}, \Psi}{\vdash [\Delta].\Gamma \leq [\Delta'].\Gamma'} \ (\textsc{subt})$$

which explicitly requests the meta implications between two state typing. By doing so, not only did we remove the fixed syntactic subtyping rules from TAL, but the State Weakening lemma is no longer part of the soundness proof. More importantly, the subtyping between preconditions is no longer limited to the built-in rules. Any valid implication relation between state typing is allowed. This is much more flexible and powerful.

Now let us look at the set of instruction typing rules ($\vdash \{\Gamma\}\mathtt{c}\{\Gamma'\}$). Also look at the Instruction Typing lemma (Lemma 2) in TAL soundness proof. Again all these syntactic rules are just used for the meta implication between two state typings in the soundness proof. (The difference this time is that the two states are now different and are states before and after the execution of an instruction.) Applying our trick again, we can replace these rules with a single one,

$$\frac{\Psi \vdash \mathbb{S} : [\Delta].\Gamma \ \supset \ \Psi \vdash \mathtt{Next_c}(\mathbb{S}) : [\Delta].\Gamma' \qquad \forall\, \mathbb{S}, \Psi}{\vdash \{\Gamma\}\mathtt{c}\{\Gamma'\}} \ (\textsc{instr}).$$

We also successfully removed the fixed syntactic instruction typing rules as well as he Instruction Typing lemma from TAL. The new form of instruction typing is much more flexible and powerful.

Of course, in composing the actual meta proof supplied to the new SUBT and INSTR rules, it is most likely that those disappeared lemmas will still be used. Neverthless, making them separated from the type language and its meta theory is important because it reduces the size of the type language, as well as allows more flexible reasoning. By introducing meta implication into TAL, we made one step forward so now there is a mixture of syntactic types and logic proofs.

We call this version of TAL a *semantic* TAL (STAL). STAL and TAL share the exactly same syntax and top-level static semantics, as well as the same state and value typing rules (The remaining of Figure 4). While STAL has much more reasoning power than TAL do, the soundness of STAL are simpler than TAL's. Nevertheless, instead of merely supplying type signatures to their code, now the programmers have to also supply actual meta logic proof.

As an intermediate step from TAL toward CAP, STAL only upgrades TAL's expressiveness by using general logic implications in the subtyping and instruction typing rules. STAL program specifications are still fullly syntactic types and thus are not as expressive as general logic predicate.

It is obvious to obtain the following TAL to STAL typing translation theorm. (we ignore the cases of those judgments where TAL and STAL share the exactly same rules.)

Theorem 2 (Typing Preservations from TAL to STAL).

1. if $\vdash_{\text{TAL}} [\Delta].\Gamma \leq [\Delta'].\Gamma'$ then $\vdash_{\text{STAL}} [\Delta].\Gamma \leq [\Delta'].\Gamma'$;
2. if $\vdash_{\text{TAL}} \{\Gamma\}\mathtt{c}\{\Gamma'\}$ then $\vdash_{\text{STAL}} \{\Gamma\}\mathtt{c}\{\Gamma'\}$.

4 Target Language: XCAP+

4.1 XCAP: A Certified Assembly Programming Language

Certified assembly programming (CAP) [15] is logic-based verification framework for assembly code. Recently, XCAP [27], a language following CAP, was proposed to solve the problem of modular support of higher-order features like embedded code pointers (ECPs) and impredicative polymorphisms. This paper uses XCAP as the base language for the target of translation from TAL.

Figure 5 defines the core of the XCAP specification language called *extended logical propositions* (*PropX*). *PropX* can be viewed as a lifted version of the meta logic propositions, extended with an cptr constant to specify ECPs. *PropX* can be used to construct assertions on machine states. For example, to specify that $\mathtt{r1}$, $\mathtt{r2}$, and $\mathtt{r3}$ store the same value, we write $\lambda(\mathbb{H},\mathbb{R}).\langle \mathbb{R}(\mathtt{r1}) = \mathbb{R}(\mathtt{r2}) \wedge \mathbb{R}(\mathtt{r_2}) = \mathbb{R}(\mathtt{r3})\rangle$. We also define code heap specifications and assertion subsumption ($\Rightarrow$) accordingly.

To establish the *validity of extended propositions*, XCAP defines a set of inductively defined validity rules in Figure 6. The judgment, $\Gamma \vdash_\Psi \mathtt{P}$, means that $\mathtt{P}$ is valid under environment Γ (a set of extended propositions) and code heap specification Ψ. The introduction rules of lifted proposition $\langle p \rangle$ and ECP proposition $\mathsf{cptr}(\mathtt{f},\mathtt{a})$ require that p and $\Psi(\mathtt{f}) = \mathtt{a}$ be valid in the meta logic. The interpretation of extended propositions is defined as their validity under the empty environment: $[\![\mathtt{P}]\!]_\Psi \triangleq \cdot \vdash_\Psi \mathtt{P}$.

We present the XCAP inference rules also in Figure 5. Comparing to TAL's top-level static semantics, we can see that they share similar structures. This is convenient for the later translation. Due to space constraint, we only explain the rules that are different from TAL.

In the top level rule, the current state is checked upon the interpretation of current assertion. For simple instructions, we require the pre- and post-condition satisfy the

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

$$\frac{\vdash \mathbb{C}:\Psi \qquad (\llbracket \mathtt{a} \rrbracket_\Psi\ \mathbb{S}) \qquad \Psi \vdash \{\mathtt{a}\}\,\mathbb{I}}{\Psi \vdash \{\mathtt{a}\}\,(\mathbb{C},\mathbb{S},\mathbb{I})}\ (\textsc{prog})$$

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

$$\frac{\Psi \vdash \{\Psi(\mathtt{f})\}\,\mathbb{C}(\mathtt{f}) \qquad \forall \mathtt{f} \in \mathsf{dom}(\Psi)}{\vdash \mathbb{C}:\Psi}\ (\textsc{cdhp})$$

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

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

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

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

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

$$\begin{array}{l}(\textit{CdHpSpec})\ \ \Psi ::= \{\mathtt{f} \rightsquigarrow \mathtt{a}\}^* \\ (\textit{Assertion})\ \ \mathtt{a} \in \textit{State} \to \textit{PropX} \\ (\textit{PropX})\ \ \mathtt{P},\mathtt{Q} ::= \langle p\rangle \mid \mathsf{cptr}(\mathtt{f},\mathtt{a}) \mid \mathtt{P} \mathbin{\mathbb{\wedge}} \mathtt{Q} \mid \mathtt{P} \mathbin{\mathbb{\vee}} \mathtt{Q} \mid \mathtt{P} \twoheadrightarrow \mathtt{Q} \mid \mathbb{\forall} x{:}A.\,\mathtt{P} \mid \mathbb{\exists} x{:}A.\,\mathtt{P} \mid \mathbb{\exists} \alpha{:}A \to \textit{PropX}.\,\mathtt{P} \\ \llbracket \mathtt{a} \rrbracket_\Psi \triangleq \lambda\mathbb{S}.\,\llbracket \mathtt{a}\ \mathbb{S} \rrbracket_\Psi \\ \mathtt{a} \Rightarrow \mathtt{a}' \triangleq \forall \Psi,\mathbb{S}.\,\llbracket \mathtt{a} \rrbracket_\Psi\ \mathbb{S} \supset \llbracket \mathtt{a}' \rrbracket_\Psi\ \mathbb{S} \\ \mathtt{a} \Rightarrow_{\mathtt{c}} \mathtt{a}' \triangleq \forall \Psi,\mathbb{S}.\,\llbracket \mathtt{a} \rrbracket_\Psi\ \mathbb{S} \supset \llbracket \mathtt{a}' \rrbracket_\Psi\ \mathtt{Next}_{\mathtt{c}}(\mathbb{S})\end{array}$$

Fig. 5. Assertion language and inference rules for XCAP

$\boxed{\Gamma \vdash_\Psi \mathtt{P}}$ ***(Validity of Extended Propositions)*** *(The following omits the Ψ in $\Gamma \vdash_\Psi P$.)*

$$(\textit{env})\ \ \Gamma ::= \cdot \mid \Gamma,\mathtt{P} \qquad \frac{\mathtt{P} \in \Gamma}{\Gamma \vdash \mathtt{P}}\ (\textsc{env}) \qquad \frac{p}{\Gamma \vdash \langle p\rangle}\ (\langle\rangle\text{-I}) \qquad \frac{\Gamma \vdash \langle p\rangle \qquad p \supset (\Gamma \vdash \mathtt{Q})}{\Gamma \vdash \mathtt{Q}}\ (\langle\rangle\text{-E})$$

$$\frac{\Psi(\mathtt{f}) = \mathtt{a}}{\Gamma \vdash \mathsf{cptr}(\mathtt{f},\mathtt{a})}\ (\textsc{cp-i}) \qquad \frac{\Gamma \vdash \mathsf{cptr}(\mathtt{f},\mathtt{a}) \quad (\Psi(\mathtt{f}) = \mathtt{a}) \supset (\Gamma \vdash \mathtt{Q})}{\Gamma \vdash \mathtt{Q}}\ (\textsc{cp-e}) \qquad \frac{\Gamma \vdash \mathtt{P} \quad \Gamma \vdash \mathtt{Q}}{\Gamma \vdash \mathtt{P} \mathbin{\mathbb{\wedge}} \mathtt{Q}}\ (\mathbb{\wedge}\text{-I})$$

$$\frac{\Gamma \vdash \mathtt{P} \mathbin{\mathbb{\wedge}} \mathtt{Q}}{\Gamma \vdash \mathtt{P}}\ (\mathbb{\wedge}\text{-E1}) \qquad \frac{\Gamma \vdash \mathtt{P} \mathbin{\mathbb{\wedge}} \mathtt{Q}}{\Gamma \vdash \mathtt{Q}}\ (\mathbb{\wedge}\text{-E2}) \qquad \frac{\Gamma \vdash \mathtt{P}}{\Gamma \vdash \mathtt{P} \mathbin{\mathbb{\vee}} \mathtt{Q}}\ (\mathbb{\vee}\text{-I1}) \qquad \frac{\Gamma \vdash \mathtt{Q}}{\Gamma \vdash \mathtt{P} \mathbin{\mathbb{\vee}} \mathtt{Q}}\ (\mathbb{\vee}\text{-I2})$$

$$\frac{\Gamma \vdash \mathtt{P} \mathbin{\mathbb{\vee}} \mathtt{Q} \quad \Gamma,\mathtt{P} \vdash \mathtt{R} \quad \Gamma,\mathtt{Q} \vdash \mathtt{R}}{\Gamma \vdash \mathtt{R}}\ (\mathbb{\vee}\text{-E}) \qquad \frac{\Gamma,\mathtt{P} \vdash \mathtt{Q}}{\Gamma \vdash \mathtt{P} \twoheadrightarrow \mathtt{Q}}\ (\twoheadrightarrow\text{-I}) \qquad \frac{\Gamma \vdash \mathtt{P} \twoheadrightarrow \mathtt{Q} \quad \Gamma \vdash \mathtt{P}}{\Gamma \vdash \mathtt{Q}}\ (\twoheadrightarrow\text{-E})$$

$$\frac{\Gamma \vdash \mathtt{P}[B/x] \quad \forall B{:}A}{\Gamma \vdash \mathbb{\forall} x{:}A.\,\mathtt{P}}\ (\mathbb{\forall}\text{-I1}) \qquad \frac{\Gamma \vdash \mathbb{\forall} x{:}A.\,\mathtt{P} \quad B{:}A}{\Gamma \vdash \mathtt{P}[B/x]}\ (\mathbb{\forall}\text{-E1}) \qquad \frac{B{:}A \quad \Gamma \vdash \mathtt{P}[B/x]}{\Gamma \vdash \mathbb{\exists} x{:}A.\,\mathtt{P}}\ (\mathbb{\exists}\text{-I1})$$

$$\frac{\Gamma \vdash \mathbb{\exists} x{:}A.\,\mathtt{P} \qquad \Gamma,\mathtt{P}[B/x] \vdash \mathtt{Q} \qquad \forall B{:}A}{\Gamma \vdash \mathtt{Q}}\ (\mathbb{\exists}\text{-E1}) \qquad \frac{\mathtt{a}{:}A \to \textit{PropX} \qquad \Gamma \vdash \mathtt{P}[\mathtt{a}/\alpha]}{\Gamma \vdash \mathbb{\exists} \alpha{:}A \to \textit{PropX}.\,\mathtt{P}}\ (\mathbb{\exists}\text{-I2})$$

Fig. 6. Validity rules for extended propositions (*PropX*)

weakest precondition relation. Direct jump is safe as long as the target code's precondition is weaker than the current one. For 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 introduce new ECP propositions about any labels in the code heap specification into the new assertion.

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

Theorem 3 (Soundness of *PropX* Interpretation).

1. If $[\![\langle p \rangle]\!]_\Psi$ then p;
2. if $[\![\mathsf{cptr}(\mathtt{f},\mathtt{a})]\!]_\Psi$ then $\Psi(\mathtt{f}) = \mathtt{a}$;
3. if $[\![\mathtt{P} \mathbb{\wedge} \mathtt{Q}]\!]_\Psi$ then $[\![\mathtt{P}]\!]_\Psi$ and $[\![\mathtt{Q}]\!]_\Psi$;
4. if $[\![\mathtt{P} \mathbb{\vee} \mathtt{Q}]\!]_\Psi$ then either $[\![\mathtt{P}]\!]_\Psi$ or $[\![\mathtt{Q}]\!]_\Psi$;
5. if $[\![\mathtt{P} \twoheadrightarrow \mathtt{Q}]\!]_\Psi$ and $[\![\mathtt{P}]\!]_\Psi$ then $[\![\mathtt{Q}]\!]_\Psi$;
6. if $[\![\mathbb{\forall} x{:}A.\,\mathtt{P}]\!]_\Psi$ and $B{:}A$ then $[\![\mathtt{P}[B/x]]\!]_\Psi$;
7. if $[\![\mathbb{\exists} x{:}A.\,\mathtt{P}]\!]_\Psi$ then there exists $B{:}A$ such that $[\![\mathtt{P}[B/x]]\!]_\Psi$;
8. if $[\![\mathbb{\exists} \alpha{:}A \rightarrow \mathit{PropX}.\,\mathtt{P}]\!]_\Psi$ then there exists $\mathtt{a}{:}A \rightarrow \mathit{PropX}$ such that $[\![\mathtt{P}[\mathtt{a}/\alpha]]\!]_\Psi$.

The soundness of XCAP is also proved in the similar way as for TAL.

Theorem 4 (XCAP Soundness). If $\Psi \vdash \{\mathtt{a}\}\,\mathbb{P}$, then for any natural number n, there exists a program $\mathbb{P}'$ such that $\mathbb{P} \longmapsto^n \mathbb{P}'$.

In the following sections, we present two extensions to XCAP. These extensions only change the assertion language and do not alter the structure of XCAP. As a result, soundness and many other meta properties of XCAP easily hold. Our extensions, including the validity soundness and XCAP soundness theorems, have been mechanized in Coq Proof Assistant [17]. Our implementation is available at [29].

4.2 Supporting Logical Recursive Specifications

Recursive specifications are very useful in describing complex invariants. Simple recursive data structures such as link-list are already supported by XCAP. However, for the recursive types ($\mu\alpha.\tau$) found in TAL, their counterpart in logic, "recursive predicates", can not be easily defined in XCAP. We add a piece of syntax to *PropX* to support recursive predicates and use syntactic methods to establish their validities.

We define the recursive predicate constructor as below.

$$(\mathit{PropX}) \quad \mathtt{P},\mathtt{Q} \quad ::= \quad \ldots \mid (\boldsymbol{\mu}\ \alpha{:}A \rightarrow \mathit{PropX}.\lambda x{:}A.\mathtt{P}\ \ B)$$

To understand the formation of recursive predicate, let us start from the innermost proposition $\mathtt{P}$. $\lambda x{:}A.\mathtt{P}$ is a predicate of type $A \rightarrow \mathit{PropX}$. So $\boldsymbol{\mu}\alpha{:}A \rightarrow \mathit{PropX}.\lambda x{:}A.\mathtt{P}$ is meant to be a recursive predicate of type $A \rightarrow \mathit{PropX}$, which corresponds to recursive types found in type systems. Since the basic unit of definition is extended proposition instead of extended predicate, we apply it with a term B of type A, and make

$(\boldsymbol{\mu}\ \alpha{:}A \to \mathit{PropX}.\lambda x{:}A.\,\mathtt{P}\ \ B)$ the basic shape of recursive predicates formula. When using recursive predicates formulas, we often use its predicate form, and use the notation $\boldsymbol{\mu}\alpha{:}A \to \mathit{PropX}.\lambda x{:}A.\,\mathtt{P}$ to represent predicate $\lambda y{:}A.(\boldsymbol{\mu}\ \alpha{:}A \to \mathit{PropX}.\lambda x{:}A.\,\mathtt{P}\ \ y)$.

To establish validity of recursive predicate formulas, we add the following rule to the validity rules in Figure 6. It is essentially a fold rule for recursive types.

$$\frac{B{:}A \qquad \Gamma \vdash \mathtt{P}[B/x][\boldsymbol{\mu}\alpha{:}A \to \mathit{PropX}.\lambda x{:}A.\,\mathtt{P}/\alpha]}{\Gamma \vdash (\boldsymbol{\mu}\ \alpha{:}A \to \mathit{PropX}.\lambda x{:}A.\,\mathtt{P}\ \ B)}\ (\mu\text{-I})$$

We extend *PropX* interpretation soundness (Theorem 3) with the following case:

If $[\![(\boldsymbol{\mu}\ \alpha{:}A \to \mathit{PropX}.\lambda x{:}A.\,\mathtt{P}\ \ B)]\!]_\Psi$ then $[\![\mathtt{P}[B/x][\boldsymbol{\mu}\alpha{:}A \to \mathit{PropX}.\lambda x{:}A.\,\mathtt{P}/\alpha]]\!]_\Psi$.

4.3 Supporting Logical mutable reference

Weak update (also termed as "mutable reference" or "general references") is a commonly used memory mutation model. Examples of weak update include ML reference cells (int ref) and managed data pointers (int _gc*) in .NET common type system. In the weak update model, each memory cell's values must satisfy a certain fixed value type. The tuple type in TAL (such as the one to be shown in next section) is also a weak update reference cell types.

Unfournately, weak update is not well-supported by CAP and other Hoare-logic-based PCC systems. The problem is due to the lack of a global data heap invariant that local assertions about heap cells can be checked upon. Existing Hoare-logic-based PCC systems either avoid supporting weak update [24, 16], limit the assertions (for reference cells) to types only [30], or resort to heavyweight techniques that require the construction of complex semantic models [26, 31]. In this section we present a weak update extension of the XCAP using the similar syntactic technique for ECPs.

We add a reference cell proposition $\mathsf{ref}(\mathtt{l},\mathtt{t})$ to the extended propositions. It associates word type $\mathtt{t}$ (a value predicate) with data label $\mathtt{l}$.

$$\begin{array}{rrcl} (\mathit{PropX}) & \mathtt{P},\mathtt{Q} & ::= & \ldots \mid \mathsf{ref}(\mathtt{l},\mathtt{t}) \\ (\mathit{WordTy}) & \mathtt{t} & \in & \mathit{Word} \to \mathit{PropX} \end{array}$$

We can use the following macro to describe a record of n cells.

$$\mathsf{record}(\mathtt{l},\mathtt{t}_1,\ldots,\mathtt{t}_n) \triangleq \mathsf{ref}(\mathtt{l},\mathtt{t}_1) \mathbin{\mathbb{\wedge}} \ldots \mathbin{\mathbb{\wedge}} \mathsf{ref}(\mathtt{l}+n-1,\mathtt{t}_n)$$

To testify the validity of reference cell propositions, in the interpretation $[\![\mathtt{P}]\!]_{\Psi,\Phi}$ we need an additional "data heap specification" parameter Φ which, similar to the code heap specification Ψ, is a partial mapping from data labels to word types.

$$(\mathit{DtHpSpec}) \qquad \Phi \quad ::= \quad \{\mathtt{l} \rightsquigarrow \mathtt{t}\}^*$$

To establish validity of $\mathsf{ref}(\mathtt{l},\mathtt{t})$, data label $\mathtt{l}$ and word type $\mathtt{t}$ need to be in Φ.

$$\frac{\Phi(\mathtt{l}) = \mathtt{t}}{\Gamma \vdash_{\Psi,\Phi} \mathsf{ref}(\mathtt{l},\mathtt{t})}\ (\text{RF-I}) \qquad \frac{\Gamma \vdash_{\Psi,\Phi} \mathsf{ref}(\mathtt{l},\mathtt{t}) \qquad (\Phi(\mathtt{l}) = \mathtt{t}) \supset (\Gamma \vdash \mathtt{Q})}{\Gamma \vdash_{\Psi,\Phi} \mathtt{Q}}\ (\text{RF-E})$$

Validity rules of other cases remain unchanged other than taking an extra Φ argument. Validity soundness of those cases also holds, with the following additional case.

$$\text{If } [\![\mathsf{ref}(\mathtt{l},\mathtt{t})]\!]_{\Psi,\Phi} \text{ then } \Phi(\mathtt{l}) = \mathtt{t};$$

Different from code heap, the data heap is dynamic. Its specification can not be obtained statically. Instead, we need to find it out in each execution steps. So the assertion interpretation is changed to:

$$[\![\mathtt{a}]\!]_{\Psi} \triangleq \lambda(\mathbb{H},\mathbb{R}).\, \exists \Phi, \mathbb{H}_s, \mathbb{H}_w.\, \mathbb{H} = \mathbb{H}_s \uplus \mathbb{H}_w \wedge [\![\mathtt{a}\ (\mathbb{H}_s,\mathbb{R})]\!]_{\Psi,\Phi} \wedge \mathcal{DH}\ \Psi\ \Phi\ \mathbb{H}_w$$

There should exist a data heap specification Φ describing the weak update part of current data heap. Other than checking validity of $(\mathtt{a}\ (\mathbb{H},\mathbb{R}))$ using Ψ and Φ, we also needs to checking validity of Φ.

For a data heap specification Φ to be valid, each reference cell must contains value that matches its word type.

$$\mathcal{DH}\ \Psi\ \Phi\ \mathbb{H} \triangleq \forall \mathtt{l} \in \mathsf{dom}(\Phi) = \mathsf{dom}(\mathbb{H}).\, [\![\Phi(\mathtt{l})\ \mathbb{H}(\mathtt{l})]\!]_{\Psi,\Phi}$$

Based on the weak update memory model defined above, we can derived many useful "macro" inference rules below. These rules can help guide the proof process by the programmer. ($\mathsf{insens}(\mathtt{a},\mathtt{r})$ asserts predicate is insensitive to register $\mathtt{r}$, *i.e.,* does not talk about register $\mathtt{r}$. Its definition is omitted here.)

$$\frac{\Psi \vdash \{(\lambda(\mathbb{H},\mathbb{R}).\, \mathtt{a}(\mathbb{H},\mathbb{R}) \mathbb{\wedge} \mathtt{t}\ \mathbb{R}(\mathtt{r}_d))\}\, \mathbb{I} \qquad \mathsf{insens}(\mathtt{a},\mathtt{r}_d)}{\Psi \vdash \{(\lambda(\mathbb{H},\mathbb{R}).\, \mathtt{a}(\mathbb{H},\mathbb{R}) \mathbb{\wedge} \mathsf{ref}(\mathbb{R}(\mathtt{r}_s)+\mathtt{w},\mathtt{t}))\}\, \mathsf{ld}\ \mathtt{r}_d, \mathtt{r}_s(\mathtt{w}); \mathbb{I}} \text{ (W-LD)}$$

$$\frac{\Psi \vdash \{\mathtt{a}\}\, \mathbb{I}}{\Psi \vdash \{(\lambda(\mathbb{H},\mathbb{R}).\, \mathtt{a}(\mathbb{H},\mathbb{R}) \mathbb{\wedge} \mathsf{ref}(\mathbb{R}(\mathtt{r}_d)+\mathtt{w},\mathtt{t}) \mathbb{\wedge} \mathtt{t}\ \mathbb{R}(\mathtt{r}_s))\}\, \mathsf{st}\ \mathtt{r}_d(\mathtt{w}), \mathtt{r}_s; \mathbb{I}} \text{ (W-ST)}$$

To avoid confusion, we call the XCAP language plus the logical recursive specification and logical mutable reference extension as the XCAP+ language, which serves as the target language of the translation.

Example Below is a "mini object", using the recursive predicates and weak update extensions presented in previous sections.

```
classs c {
  void f (c x) { x.f(x) }
}
```

$$c \triangleq \boldsymbol{\mu}\ \alpha{:}\mathit{Word} \to \mathit{PropX}.\, \lambda x{:}\mathit{Word}.\, \mathsf{ref}(x,\ \lambda y{:}\mathit{Word}.\, \mathsf{cptr}(y,\ \lambda(\mathbb{H},\mathbb{R}).\, (\alpha\ \mathbb{R}(\mathtt{r1}))))$$

The above example may looks similar to what one would normally write in a syntactic type system. However, given the ability to write general logic predicate in the sepcifications, it is possible to compose very interesting data structures and properties. Below is an example: a record pointer $\mathtt{l}$ which points to an even number, a data pointer to a reference cell storing an odd number, and a code pointer which, among other things, expects register $\mathtt{r}_1$ to be an unaliased pointer to a prime number. ($\mapsto$ and $*$ are separation logic connectives which can be build in XCAP using shallow embedding.)

$$\mathsf{record}(\mathtt{l},\ \mathsf{even},\ \lambda \mathtt{w}.\, \mathsf{ref}(\mathtt{w},\mathsf{odd}),\ \lambda \mathtt{w}.\, \mathsf{cptr}(\mathtt{w},\ \lambda(\mathbb{H},\mathbb{R}).\, (\exists \mathtt{w}.\, \mathbb{R}(\mathtt{r}_1) \mapsto \mathtt{w} \wedge \mathsf{prime}\ \mathtt{w}) * \ldots))$$

Translation from TAL WordTy to XCAP+ WordTy

$$\begin{aligned}
\ulcorner \mathsf{int} \urcorner &\triangleq \lambda \mathtt{w}.\,\mathsf{True} \\
\ulcorner \mathsf{code}[\Delta].\Gamma \urcorner &\triangleq \lambda \mathtt{w}.\,\mathsf{codeptr}(\mathtt{w}, \ulcorner [\Delta].\Gamma \urcorner) \\
\ulcorner \langle \tau_1, \ldots, \tau_n \rangle \urcorner &\triangleq \lambda \mathtt{w}.\,\mathsf{record}(\mathtt{w}, \ulcorner \tau_1 \urcorner, \ldots, \ulcorner \tau_n \urcorner) \\
\ulcorner \exists \alpha.\tau \urcorner &\triangleq \lambda \mathtt{w}.\,\exists \alpha : \mathit{Word} \to \mathit{PropX}.\ulcorner \tau \urcorner \\
\ulcorner \mu \alpha.\tau \urcorner &\triangleq \lambda \mathtt{w}.\,(\boldsymbol{\mu}\ \alpha : \mathit{Word} \to \mathit{PropX}.\lambda x : \mathit{Word}.\,(\ulcorner \tau \urcorner\ x)\ \ \mathtt{w})
\end{aligned}$$

Translation from TAL Precondition to XCAP+ Assertion

$$\begin{aligned}
&\ulcorner [\alpha_1, \ldots, \alpha_m].\{\mathtt{r}_1 \rightsquigarrow \tau_1, \ldots, \mathtt{r}_n \rightsquigarrow \tau_n\} \urcorner \\
&\quad \triangleq\ \lambda(\mathbb{H}, \mathbb{R}).\,\exists \alpha_1, \ldots, \alpha_m : \mathit{Word} \to \mathit{PropX}.\,(\ulcorner \tau_1 \urcorner\ \mathbb{R}(\mathtt{r}_1)) \mathbin{/\!\!\backslash} \ldots \mathbin{/\!\!\backslash} (\ulcorner \tau_n \urcorner\ \mathbb{R}(\mathtt{r}_n))
\end{aligned}$$

Translation from TAL CdHpSpec to XCAP+ CdHpSpec

$$\ulcorner \{\mathtt{l}_1 \rightsquigarrow [\Delta_1].\Gamma_1, \ldots, \mathtt{l}_n \rightsquigarrow [\Delta_n].\Gamma_n\} \urcorner \ \triangleq\ \{\mathtt{l}_1 \rightsquigarrow \ulcorner [\Delta_1].\Gamma_1 \urcorner, \ldots, \mathtt{l}_n \rightsquigarrow \ulcorner [\Delta_n].\Gamma_n \urcorner\}$$

Translation from TAL DtHpSpec to XCAP+ DtHpSpec

$$\ulcorner \{\mathtt{l}_1 \rightsquigarrow \tau_1, \ldots, \mathtt{l}_n \rightsquigarrow \tau_n\} \urcorner \ \triangleq\ \{\mathtt{l}_1 \rightsquigarrow \ulcorner \tau_1 \urcorner, \ldots, \mathtt{l}_n \rightsquigarrow \ulcorner \tau_n \urcorner\}$$

Fig. 7. Type translations from TAL/STAL to XCAP+

5 A Translation from TAL/STAL to XCAP+

We first give a translation from TAL/STAL types to XCAP+ assertions. We then show the type-preserving property of the STAL to XCAP+ translation. Finally we do some brief discussion of the translation.

5.1 Type Translations from TAL/STAL to XCAP+

We present the type translations from TAL to XCAP+ in Figure 7. Various $\ulcorner \cdot \urcorner$ translate TAL types into XCAP+ assertions and specifications.

In the word type translation, integer type becomes a tautology as any machine word can be treated as an integer. Code type is translated into ECP formulas. Tuple types in TAL is translated into record type in XCAP+. Existential types and recursive types in TAL are also translated into their XCAP+ counterparts.

The translation of a TAL precondition, which is a type variable environment plus a register file type, is an XCAP+ assertions. The type variables in the environment are now existentially quantified over XCAP+ word types at the outmost of the target assertion. The register file typing corresponds to a bunch of conjunction of register value checking.

The translations of TAL code and data heap specifications are carried out by simply translating each element's type in them into XCAP+ assertions or word types, and preserving the partial mapping.

5.2 Typing Preservations from TAL/STAL to XCAP+

An important property of the previous type translations is whether they preserve the typing, *i.e.*, whether well-formed TAL/STAL entities are still well-formed in XCAP+ after the translation. As the following typing preservation lemma show, all STAL typing derivations gets preserved in XCAP+ after the translations.

Theorem 5 (Typing Preservations from STAL to XCAP+).

1. If $\Psi \vdash_{\text{STAL}} \{[\Delta].\Gamma\}\,\mathbb{P}$ then $\ulcorner\Psi\urcorner \vdash_{\text{XCAP+}} \{\ulcorner[\Delta].\Gamma\urcorner\}\,\mathbb{P}$;
2. if $\vdash_{\text{STAL}} \mathbb{C} : \Psi$ then $\vdash_{\text{XCAP+}} \mathbb{C} : \ulcorner\Psi\urcorner$;
3. if $\Psi \vdash_{\text{STAL}} \{[\Delta].\Gamma\}\mathbb{I}$ then $\ulcorner\Psi\urcorner \vdash_{\text{XCAP+}} \{\ulcorner[\Delta].\Gamma\urcorner\}\mathbb{I}$;
4. if $\Psi \vdash_{\text{STAL}} \mathbb{S} : [\Delta].\Gamma$ then $[\![\ulcorner[\Delta].\Gamma\urcorner]\!]_{\ulcorner\Psi\urcorner}\,\mathbb{S}$;
5. if $\Psi \vdash_{\text{STAL}} \mathbb{H} : \Phi$ then $\mathcal{DH}\ \ulcorner\Psi\urcorner\ \ulcorner\Phi\urcorner\,\mathbb{H}$;
6. if $\Psi;\Phi \vdash_{\text{STAL}} \mathbb{R} : \Gamma$ then $[\![\ulcorner[].\Gamma\urcorner\ (\mathbb{H},\mathbb{R})\,]\!]_{\ulcorner\Psi\urcorner,\ulcorner\Phi\urcorner}$;
7. if $\Psi;\Phi \vdash_{\text{STAL}} \mathtt{w} : \tau$ then $[\![\ulcorner\tau\urcorner\ \mathtt{w}]\!]_{\ulcorner\Psi\urcorner,\ulcorner\Phi\urcorner}$;
8. if $\vdash_{\text{STAL}} [\Delta].\Gamma \leq [\Delta'].\Gamma'$ then $\ulcorner[\Delta].\Gamma\urcorner \Rightarrow \ulcorner[\Delta'].\Gamma'\urcorner$;
9. if $\vdash_{\text{STAL}} \{\Gamma\}\mathtt{c}\{\Gamma'\}$ then $\ulcorner[\Delta].\Gamma\urcorner_{\Psi} \Rightarrow_{\mathtt{c}} \ulcorner[\Delta].\Gamma'\urcorner$.

Proof. We show selected cases for (3). By induction over the structure of $\Psi \vdash_{\text{TAL}} \{[\Delta].\Gamma\}\mathbb{I}$.

case WEAKEN. $$\frac{\vdash [\Delta].\Gamma \leq [\Delta'].\Gamma' \qquad \Psi \vdash \{[\Delta'].\Gamma'\}\,\mathbb{I}}{\Psi \vdash \{[\Delta].\Gamma\}\,\mathbb{I}}$$

By induction hypothesis it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP+}} \{\ulcorner[\Delta'].\Gamma'\urcorner\}\mathbb{I}$. The following weakening lemma easily holds for XCAP+: "if $\Psi \vdash \{\mathtt{a}'\}\,\mathbb{I}$ and $\mathtt{a} \Rightarrow \mathtt{a}'$ then $\Psi \vdash \{\mathtt{a}\}\,\mathbb{I}$". By (8) it follows that $\ulcorner[\Delta].\Gamma\urcorner \Rightarrow \ulcorner[\Delta'].\Gamma'\urcorner$. Thus it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP+}} \{\ulcorner[\Delta].\Gamma\urcorner\}\mathbb{I}$.

case SEQ. $$\frac{\Psi \vdash \{\Gamma\}\mathtt{c}\{\Gamma'\} \qquad \Psi \vdash \{[\Delta].\Gamma'\}\,\mathbb{I} \qquad \mathtt{c} \in \{\mathsf{add},\mathsf{mov},\mathsf{movi},\mathsf{ld},\mathsf{st}\}}{\Psi \vdash \{[\Delta].\Gamma\}\,\mathtt{c};\mathbb{I}}$$

By induction hypothesis it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP+}} \{\ulcorner[\Delta].\Gamma'\urcorner\}\mathbb{I}$. By (9) it follows that $\ulcorner[\Delta].\Gamma\urcorner_{\Psi} \Rightarrow_{\mathtt{c}} \ulcorner[\Delta].\Gamma'\urcorner$. By rule SEQ it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP+}} \{\ulcorner[\Delta].\Gamma\urcorner\}\mathtt{c};\mathbb{I}$.

case JD. $$\frac{\mathtt{f} \in \mathsf{dom}(\Psi) \qquad \vdash [\Delta].\Gamma \leq \Psi(\mathtt{f})}{\Psi \vdash \{[\Delta].\Gamma\}\,\mathsf{jd}\ \mathtt{f}}$$

By (8) it follows that $\ulcorner[\Delta].\Gamma\urcorner \Rightarrow \ulcorner\Psi(\mathtt{f})\urcorner$, and then $\ulcorner[\Delta].\Gamma\urcorner \Rightarrow \ulcorner\Psi\urcorner(\mathtt{f})$. By rule JD it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP+}} \{\ulcorner[\Delta].\Gamma\urcorner\}\mathsf{jd}\ \mathtt{f}$.

case JMP. $$\frac{\Gamma(\mathtt{r}) = \mathsf{code}[\Delta'].\Gamma' \qquad \vdash [\Delta].\Gamma \leq [\Delta'].\Gamma'}{\Psi \vdash \{[\Delta].\Gamma\}\,\mathsf{jmp}\ \mathtt{r}}$$

By translation it follows that $\ulcorner[\Delta].\Gamma\urcorner \Rightarrow \lambda(\mathbb{H},\mathbb{R}).\,\mathsf{cptr}(\mathbb{R}(\mathtt{r}),\ulcorner[\Delta'].\Gamma'\urcorner)$. By (8) it follows that $\ulcorner[\Delta].\Gamma\urcorner \Rightarrow \ulcorner[\Delta'].\Gamma'\urcorner$. By rule JMP it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP+}} \{\ulcorner[\Delta].\Gamma\urcorner\}\mathsf{jmp}\ \mathtt{r}$.

case MOVF. $$\frac{\mathtt{f} \in \mathsf{dom}(\Psi) \qquad \Psi \vdash \{[\Delta].\Gamma\{\mathtt{r}_d \rightsquigarrow \mathsf{code}\ \Psi(\mathtt{f})\}\}\,\mathbb{I}}{\Psi \vdash \{[\Delta].\Gamma\}\,\mathsf{movi}\ \mathtt{r}_d,\mathtt{f};\mathbb{I}}$$

By induction hypothesis it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP+}} \{\ulcorner[\Delta].\Gamma\{\mathtt{r}_d \rightsquigarrow \mathsf{code}\ \Psi(\mathtt{f})\}\urcorner\}\mathbb{I}$. By rule ECP and SEQ it follows that $\ulcorner\Psi\urcorner \vdash_{\text{XCAP+}} \{\ulcorner[\Delta].\Gamma\urcorner\}\mathsf{movi}\ \mathtt{r}_d,\mathtt{f};\mathbb{I}$. ■

5.3 Discussion

We discussed the relationship between TAL and CAP by showing a translation from TAL to XCAP+. However, this work is by no means limited to the translation itself.

For example, one can treat the translation as a shallow embeddeding of TAL types and typing rules in XCAP+, which means there is no need to build meta theory for TAL while still letting the programmer working at TAL level.

One interesting perspective to view the XCAP+ system is from TAL programmer's side. When writting programs that will interact with XCAP+ code that involve complex logical specification not expressible in TAL, so long as the interfaces abstract and hide that part of "real" logical specification (e.g., by using existential packages), and local code behavior is simple enough (which is usually the case), the programmers can "pretend" that they are dealing with external TAL code instead of XCAP+ ones, by using the macros and lemmas defined for the translations. This lowers the requirement on programmers, as they do not need to learn XCAP+, even if they are dealing with external XCAP+ code.

6 Related Work and Conclusion

Hamid and Shao [30] introduced an Extensible TAL with Runtime System (XTAL) and showed its translation to an earlier version of Certified Assembly Programming. Their main goal is to make sure the translation result can interoperate with the run-time system written and verified in CAP and thus form a complete FPCC proof. Since CAP does not have modular support of embedded code pointer and weak update, the translation there can not be merely from XTAL value types to CAP predicates, but rather needs XTAL heap types as well. This requires both the source type systems and programs to be fixed.

Chen *et al* [4] have developed LTAL, a low-level typed assembly language which is used to compile core ML to FPCC. LTAL is based in turn upon an abstraction layer, TML (typed machine language) [32], which is an even lower-level intermediate language. Complex parts of the semantic proofs, such as the indexed model of recursive types and stratified model of mutable fields, are hidden in the soundness proof of TML.

Recently, Appel *et al* [33] proposed a new modal model for expressing recursive and impredicatively quantified types with mutable reference. Their method uses a Kripke semantics of the Godel-Lob logic of provability. To support mutable reference, they include the memory mapping information in the "world". A full comparison between the CAP/XCAP line of work and the line of index-based semantic FPCC work will be an interesting yet non-trivial problem. Here we are only going to list a few of them.

First, CAP is designed to support more than "type safety" at each program point. The code heap specification specifies the safety policy in general logic predicates, and is completely customizable. Meta theory of CAP guarantees that these safety policy is actually observable. While in indexed FPCC work, the definition of "codeptr" has always been based on a non-stuckness safety. It remains unclear how to change the definition of "codeptr" in order to support customizable and observable safety policies without rebuilding the meta theory.

Second, CAP is designed to support directly verifying assembly programs with non-trivial properties not expressible in traditional types. Various examples of CAP programs has been mechanically certified [22, 16, 18, 27]. While the indexed FPCC work has been focusing on type-preserving compilations from high-level languages, thus have few application of direct verification at assembly level. The translation presented in this paper is one effort on connecting CAP/XCAP with high-level languages.

Conclusion We presented a type-preserving translation from a TAL language to a CAP language. Our translation supports polymorphic code, mutable reference, existential, and recursive types. To do so, we used an intermediate "semantic" TAL language. We also extended the XCAP language to support logical recursive specifications and mutable references. Our result will lead to linking and inter-operating well-typed programs from traditional certifying compilers with certified libraries by CAP-like systems.

References

1. Necula, G.: Proof-carrying code. In: Proc. 24th ACM Symposium on Principles of Programming Languages, New York, ACM Press (1997) 106–119
2. Colby, C., Lee, P., Necula, G., Blau, F., Plesko, M., Cline, K.: A certifying compiler for Java. In: Proc. 2000 ACM Conference on Programming Language Design and Implementation, New York, ACM Press (2000) 95–107
3. Hamid, N.A., Shao, Z., Trifonov, V., Monnier, S., Ni, Z.: A syntactic approach to foundational proof-carrying code. In: Proc. 17th Annual IEEE Symposium on Logic in Computer Science. (2002) 89–100
4. Chen, J., Wu, D., Appel, A.W., Fang, H.: A provably sound tal for back-end optimization. In: Proc. 2003 ACM Conference on Programming Language Design and Implementation, ACM Press (2003) 208–219
5. Crary, K.: Toward a foundational typed assembly language. In: Proc. 30th ACM Symposium on Principles of Programming Languages. (2003) 198
6. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Addison-Wesley (1996)
7. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (Revised). MIT Press, Cambridge, Massachusetts (1997)
8. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. In: Proc. 25th ACM Symposium on Principles of Programming Languages, ACM Press (1998) 85–97
9. Xi, H., Pfenning, F.: Dependent types in practical programming. In: Proc. 26th ACM Symposium on Principles of Programming Languages, ACM Press (1999) 214–227
10. Shao, Z., Saha, B., Trifonov, V., Papaspyrou, N.: A type system for certified binaries. In: Proc. 29th ACM Symposium on Principles of Programming Languages, ACM Press (2002) 217–232
11. Crary, K., Vanderwaart, J.C.: An expressive, scalable type theory for certified code. In: Proc. 7th ACM SIGPLAN International Conference on Functional Programming. (2002) 191–205
12. Ahmed, A., Walker, D.: The logical approach to stack typing. In: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation, ACM Press (2003) 74–85
13. Ahmed, A., Jia, L., Walker, D.: Reasoning about hierarchical storage. In: Proc. 18th IEEE Symposium on Logic in Computer Science. (2003) 33–44

14. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: Proc. 17th Annual IEEE Symposium on Logic in Computer Science. (2002)
15. Yu, D., Hamid, N.A., Shao, Z.: Building certified libraries for PCC: Dynamic storage allocation. Science of Computer Programming **50**(1-3) (2004) 101–127
16. Yu, D., Shao, Z.: Verification of safety properties for concurrent assembly code. In: Proc. 2004 International Conference on Functional Programming (ICFP'04). (2004)
17. The Coq Development Team: The Coq proof assistant reference manual. The Coq release v8.0 (2004)
18. Feng, X., Shao, Z.: Modular verification of concurrent assembly code with dynamic thread creation and termination. In: Proc. 2005 International Conference on Functional Programming (ICFP'04). (2005)
19. Floyd, R.W.: Assigning meaning to programs. Communications of the ACM (1967)
20. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM (1969)
21. Appel, A.W.: Foundational proof-carrying code. In: Proc. 16th Annual IEEE Symposium on Logic in Computer Science. (2001) 247–258
22. Yu, D., Hamid, N.A., Shao, Z.: Building certified libraries for PCC: Dynamic storage allocation. In: Proc. 2003 European Symposium on Programming (ESOP'03). (2003)
23. Gordon, M.: A mechanized Hoare logic of state transitions. In Roscoe, A.W., ed.: A Classical Mind—Essays in Honour of C.A.R. Hoare, Prentice Hall (1994) 143–160
24. Necula, G.: Compiling with Proofs. PhD thesis, School of Computer Science, Carnegie Mellon Univ. (1998)
25. Appel, A.W., Felty, A.P.: A semantic model of types and machine instructions for proof-carrying code. In: Proc. 27th ACM Symposium on Principles of Programming Languages. (2000) 243–253
26. Appel, A.W., McAllester, D.: An indexed model of recursive types for foundational proof-carrying code. ACM Transactions on Programming Languages and Systems **23**(5) (2001) 657–683
27. Ni, Z., Shao, Z.: Certified assembly programming with embedded code pointers. In: Proc. 33rd ACM Symposium on Principles of Programming Languages. (2006)
28. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115**(1) (1994) 38–94
29. Ni, Z., Shao, Z.: Implementation for a translation from typed assembly language to certified assembly programming. `http://flint.cs.yale.edu/publications/talcap.html` (2006)
30. Hamid, N.A., Shao, Z.: Interfacing hoare logic and type systems for foundational proof-carrying code. In: Proc. 17th International Conference on Theorem Proving in Higher Order Logics. Volume 3223 of LNCS., Springer-Verlag (2004) 118–135
31. Ahmed, A.J.: Semantics of Types for Mutable State. PhD thesis, Princeton University (2004)
32. Swadi, K.N., Appel, A.W.: Typed machine language and its semantics. Unpublished manuscript available at `www.cs.princeton.edu/~appel/papers` (2001)
33. Appel, A.W., Mellies, P.A., Richards, C.D., Vouillon., J.: A very modal model of a modern, major, general type system. In: Proc. 34th ACM Symposium on Principles of Programming Languages. (2007)