Typed Assembly for the Zarf ISA

Machine-Checked, Typed, Polymorphic, Functional Assembly Code

Michael Christensen, Joseph McMahan,
Ben Hardekopf, Tim Sherwood

This Talk

- Unpublished
- Thoughts on extensions to our prior work
- Seeing if anyone else cares
- Request for:
  - Feedback
  - Experiences
  - Related work to consider
- Get us thinking about applying PL techniques to the **architecture world**

Outline

- This Talk
- The Zarf Architecture
  - Untyped Machine
  - Semantics
- Typed Assembly Background
  - Advantages
- Typed Zarf
  - *Typed* Machine
  - Binary Type-checking Algorithm
  - Some Guarantees Zarf Gets
- The Future, and Why
  - Finding Motivation

The Zarf Architecture for Recursive Functions (ASPLOS 2017)

- An ISA based on the lambda calculus
  - Lambda-lifted, lazy, ANF, WHNF
  - 3 instructions
    - **let**: stores a thunk
    - **case**: thunk evaluation to WHNF for matching
    - **result:** yields value to **case**
      - terminates every instruction branch and function
  - Functions
    - User-defined
    - Builtins (**add, mult**, **xor,** etc.)
    - I/O: **getint** and **putint**
  - Constructors
    - Named tuples
    - Special **error** constructor to indicate HW error

Untyped Machine

	High-Level Assembly	Machine Assembly	Binary			
1	**cons** nil	**cons** 0 # nil, 0x101	1	2	X	X
2	**cons** cons head tail	**cons** 2 # cons, 0x102	1	0	X	X
3	**func** map f l =	**func** 2 3: # map, 0x103	0	2	X	3
4	case l of {	case [arg 1]	1	X	0	1
5	nil =>	pat_con [0x101] 1	4	1	X	101
6	l	result [arg 1]	2	X	0	1
7	cons h tl =>	pat_con [0x102] 9	4	9	X	102
8	let h’ = f h in	let [arg 0] 1 # local 0	0	1	0	0
9		[field 0]	3		0	
10	let tl’ = map f tl in	let [table 0x103] 2 # local 1	0	2	4	103
11		[arg 0]	0		0	
12		[field 1]	3		1	
13	let l’ = cons h’ tl’ in	let [table 0x102] 2 # local 2	0	2	7	102
14		[local 0]	1		0	
15		[local 1]	1		1	
16	l’	result [local 2]	1	X	1	2
17		pat_else 2	5	2	X	X
18		let [table 0x0] 0 # local 0	0	0	4	0
19		result [local 0]	2	X	1	0

Binary Encoding

Function Header

isCons	nArgs	reserved	nLocals
1	11	10	10

Instruction Word

op	n	src	index
3	10	3	16

Argument Word

src	index
3	29

Opcodes

0	let
1	case
2	result
3	pat_lit
4	pat_con
5	pat_else

Sources

0	arg
1	local
2	literal
3	field
4	table

Semantics

$c \in Constructor = Name \times \overrightarrow{Value}$ $\quad clo \in Closure = (\lambda\vec{x}.e) \times \overrightarrow{Value}$ $\quad v \in Value = \mathbb{Z} \uplus Constructor \uplus Closure$ $\quad \rho \in Env = Variable \to Value$

$x \in Variable \quad n \in \mathbb{Z} \quad fn, cn \in Name \quad \oplus \in PrimOp$

$$
\begin{aligned}
p \in Program &::= \overrightarrow{decl}\ \textbf{fun main = } e \\
decl \in Declaration &::= cons \mid func \\
cons \in Constructor &::= \textbf{con}\ cn\ \vec{x} \\
func \in Function &::= \textbf{fun}\ fn\ \vec{x}\ \textbf{=}\ e \\
e \in Expression &::= let \mid case \mid res \\
let \in Let &::= \textbf{let}\ x\ \textbf{=}\ id\ \overrightarrow{arg}\ \textbf{in}\ e \\
case \in Case &::= \textbf{case}\ arg\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e \\
res \in Result &::= \textbf{result}\ arg \\
br \in Branch &::= cn\ \vec{x} \Rightarrow e \mid n \Rightarrow e \\
id \in Identifier &::= x \mid fn \mid cn \mid \oplus \\
arg \in Argument &::= n \mid x
\end{aligned}
$$

$$\frac{\vdash e \Downarrow v}{\overrightarrow{decl}\ \textbf{fun main = } e \Downarrow v}\ (\text{PROGRAM}) \qquad \frac{v = \rho(arg)}{\rho \vdash \textbf{result}\ arg \Downarrow v}\ (\text{RESULT})$$

$$\frac{\vec{v}_1 = \rho(\overrightarrow{arg}) \quad v_2 = \texttt{applyCn}(cn, \vec{v}_1) \quad \rho[x \mapsto v_2] \vdash e \Downarrow v_3}{\rho \vdash \textbf{let}\ x\ \textbf{=}\ cn\ \overrightarrow{arg}\ \textbf{in}\ e \Downarrow v_3}\ (\text{LET-CON})$$

$$\frac{fn \notin \{\textbf{getint}, \textbf{putint}\} \quad \textbf{fun}\ fn\ \vec{x}_2\ \textbf{=}\ e_2 \in \overrightarrow{decl} \quad \vec{v}_1 = \rho(\overrightarrow{arg}) \quad v_2 = \texttt{applyFn}((\lambda\vec{x}_2.e_2, []), \vec{v}_1, \rho) \quad \rho[x_1 \mapsto v_2] \vdash e_1 \Downarrow v_3}{\rho \vdash \textbf{let}\ x_1\ \textbf{=}\ fn\ \overrightarrow{arg}\ \textbf{in}\ e_1 \Downarrow v_3}\ (\text{LET-FUN})$$

$$\frac{= \rho(x_2) \quad \vec{v}_2 = \rho(\overrightarrow{arg}) \quad v_3 = \texttt{applyFn}(v_1, \vec{v}_2, \rho) \quad \rho[x_1 \mapsto v_3] \vdash e \Downarrow v_4}{\rho \vdash \textbf{let}\ x_1\ \textbf{=}\ x_2\ \overrightarrow{arg}\ \textbf{in}\ e \Downarrow v_4}\ (\text{LET-VAR}) \qquad \frac{n_2 \text{ is input from port } n_1 \quad \rho[x \mapsto n_2] \vdash e \Downarrow v}{\rho \vdash \textbf{let}\ x\ \textbf{= getint}\ n_1\ \textbf{in}\ e \Downarrow v}\ (\text{GETINT})$$

$$\frac{\vec{v}_1 = \rho(\overrightarrow{arg}) \quad v_2 = \texttt{applyPrim}(\oplus, \vec{v}_1) \quad \rho[x \mapsto v_2] \vdash e \Downarrow v_3}{\rho \vdash \textbf{let}\ x\ \textbf{=}\ \oplus\ \overrightarrow{arg}\ \textbf{in}\ e \Downarrow v_3}\ (\text{LET-PRIM}) \qquad \frac{n_2 = \rho(arg) \quad \rho[x \mapsto n_2] \vdash e \Downarrow v}{\rho \vdash \textbf{let}\ x\ \textbf{= putint}\ n_1\ arg\ \textbf{in}\ e \Downarrow v}\ (\text{PUTINT})$$

$$\frac{(cn, \vec{v}_1) = \rho(arg) \quad (cn\ \vec{x} \Rightarrow e_1) \in \overrightarrow{br} \quad \rho[\vec{x} \mapsto \vec{v}_1] \vdash e_1 \Downarrow v_2}{\rho \vdash \textbf{case}\ arg\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_2 \Downarrow v_2}\ (\text{CASE-CON}) \qquad \frac{n = \rho(arg) \quad (n \Rightarrow e_1) \in \overrightarrow{br} \quad \rho \vdash e_1 \Downarrow v}{\rho \vdash \textbf{case}\ arg\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_2 \Downarrow v}\ (\text{CASE-LIT})$$

$$\frac{(cn, \vec{v_1}) = \rho(arg) \quad (cn\ \vec{x} \Rightarrow e_1) \notin \overrightarrow{br} \quad \rho \vdash e_2 \Downarrow v_2}{\rho \vdash \textbf{case}\ arg\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_2 \Downarrow v_2}\ (\text{CASE-ELSE1}) \qquad \frac{n = \rho(arg) \quad (n \Rightarrow e_1) \notin \overrightarrow{br} \quad \rho \vdash e_2 \Downarrow v_1}{\rho \vdash \textbf{case}\ arg\ \textbf{of}\ \overrightarrow{br}\ \textbf{else}\ e_2 \Downarrow v_1}\ (\text{CASE-ELSE2})$$

$$\texttt{applyFn}((\lambda\vec{x}_1.e, \vec{v}_1), \vec{v}_2, \rho) = \begin{cases} v & \text{if } |\vec{v}_2| = 0,\ |\vec{v}_1| = |\vec{x}_1|, \text{ and } \rho[\vec{x}_1 \mapsto \vec{v}_1] \vdash e \Downarrow v \\ (\lambda\vec{x}_1.e, \vec{v}_1) & \text{if } |\vec{v}_2| = 0 \text{ and } |\vec{v}_1| < |\vec{x}_1| \\ \texttt{applyFn}((\lambda\vec{x}_1.e, \vec{v}_1 :+ \texttt{hd}(\vec{v}_2)), \texttt{tl}(\vec{v}_2), \rho) & \text{if } |\vec{v}_2| > 0 \text{ and } |\vec{v}_1| < |\vec{x}_1| \\ \texttt{applyFn}((\lambda\vec{x}_2.e', \vec{v}_3), \vec{v}_2, \rho) & \text{if } |\vec{v}_2| > 0,\ |\vec{x}_1| = |\vec{v}_1|, \text{ and } \rho[\vec{x}_1 \mapsto \vec{v}_1] \vdash e \Downarrow (\lambda\vec{x}_2.e', \vec{v}_3) \end{cases}$$

$$\texttt{applyCn}(cn, \vec{v}) = \begin{cases} (cn, \vec{v}) & \text{if } (\textbf{con}\ cn\ \vec{x}) \in \overrightarrow{decl} \text{ and } |\vec{v}| = |\vec{x}| \\ (\lambda\vec{x}.\textbf{let}\ c\ \textbf{=}\ cn\ \vec{x}\ \textbf{in result}\ c, \vec{v}) & \text{if } (\textbf{con}\ cn\ \vec{x}) \in \overrightarrow{decl} \text{ and } |\vec{v}| < |\vec{x}| \end{cases} \qquad \rho(arg) = \begin{cases} n & \text{if } arg = n \\ v & \text{if } arg = x \text{ and } (x \mapsto v) \in \rho \end{cases}$$

Typed Assembly Background

TAL/TALx86 (Morrisett et al. 1999, Crary et al. 1999):

- High-level lang. compilation target
- Abstraction raised
  - word → integer, pointer, tuple, code labels
- Type constructors, parametric poly.
- Preconditions on code labels (register types)

STAL (Morrisett et al. 2002)

- Extend TAL with control stack

FunTAL (Patterson et al. 2017)

- Embed assembly in typed functional language and vice versa
- Reason about TAL components (mult. BBs) and high-level exp

Also:

- PCC (Necula and Lee, 1996)
  - Agent-supplied data/proof that it code complies with host’s safety policies
- Typed intermediate languages

Typed Assembly Advantages

- High-level abstractions
  - enforced at machine level
  - can be ensured to be compiled/maintained correctly
- Help optimizations during entire compilation process
- Check untrusted code before running
  - Write in any language as long as it compiles down to TAL
- FunTAL: include speedy low-level operations and maintain guarantees

Typed Machine

#	High-Level Assembly	Machine Assembly	Binary
1	**data** List[a]	**data** 1 2 # List[a] 0x101	1 1 2
2	= Nil	**cons** 0 # Nil 0x101	1 0
3	| Cons a List[a]	**cons** 2 # Cons 0x102	1 2
4		[tvar 0]	3 0
5		[data 0x101]	1 0x101
6		[tvar 0]	3 0
7	**func** map[a,b]:	**func** 2 2 # map 0x103	0 2 2
8	((a → b),	[func 1]	2 1
9		[tvar 0]	3 0
10		[tvar 1]	3 1
11	List[a])	[data 0x101]	1 0x101
12		[tvar 0]	3 0
13	→ List[b]	[data 0x101]	1 0x101
14		[tvar 1]	3 1
15	**func** map f l =	**def** 0x103 3 # definition	0 0x103 3
16	case l of {	case [arg 1]	1 X 0 1
17	Nil =>	pat_con [0x101] 1	4 2 X 0x101
18	let n = Nil in	let [table 0x101] 0 # local 0	0 0 4 0x101
19	n	result [local 0]	2 X 1 0
20	Cons h tl =>	pat_con [0x102] 9	4 9 X 0x102
21	let h’ = f h in	let [arg 0] 1 # local 0	0 1 0 0
22		[field 0]	3 0
23	let tl’ = map f tl in	let [table 0x103] 2 # local 1	0 2 4 0x103
24		[arg 0]	0 0
25		[field 1]	3 1
26	let l’ = Cons h’ tl’ in	let [table 0x102] 2 # local 2	0 2 4 0x102
27		[local 0]	1 0
28		[local 1]	1 1
29	l’	result [local 2]	2 X 1 2

Binary Encoding

Data Declaration

isData	nTvars	nCons/nArgs
1	15	16

Table Declaration (Cons)

isCons	nFields
1	31

Table Declaration (Func)

isCons	index	nLocals
1	16	15

Type Word

type	index/nArgs
2	30

Instruction Word

op	nArgs	src	index
3	10	3	16

Argument Word

src	index
3	29

Types

0	int	2	func
1	data	3	tvar

Opcodes

0	let
1	case
2	result
3	pat_lit
4	pat_con
5	pat_else

Sources

0	arg
1	local
2	literal
3	field
4	table

Binary Type Checking Algorithm

1. Load argument and return types with type variables
2. For each instruction:
    a. If **let**:
        i. Lookup table type
            1. Load its expected arguments/fields, with fresh, consistent type variable indices
        ii. Lookup argument types in environment (must be variables or primitives)
        iii. Check types:
            1. Match concrete type constructor
            2. Add constraints between type variables
    b. If **case**:
        i. Lookup scrutinee type in environment
        ii. Lookup data type (if not primitive)
            1. Load constructors, assign type of fields to types in scrutinee
        iii. Check for totality of matching
    c. If **result:**
        i. Unify constraints, replace non-original type variables as needed
        ii. Compare result to return type

Some Guarantees Zarf Gets

- Filters our bad behaviors
  - Casing on an underapplied thunk/constructor
  - Passing incorrect arguments, using return value incorrectly
  - Elimination of *some* runtime error constructors (still array-out-of-bounds possibilities)
- Removes the need for a compiler you totally trust
- All foreign untrusted binaries can be checked
  - Untyped Zarf already prohibits arbitrary control flow and memory access...
- It's lambda-calculus (basically), so verifying it should be eas(ier)
- Type-checking done in **HW**; cannot be circumvented
  - no variable length memory structures
  - fixed bit width
  - bounded state machine

Finding Motivation

- Why do we need another typed assembly?
- Is there a meaningful decrease in proof complexity by using Zarf?
- Is hardware type-checking feasible for a non-trivial type system?
- Is our type system expressive enough?
- What kind of type system do *you* want?
- What kind of errors do you want to prevent?
- Ideas and future work:
    - Implement on hardware (currently in our simulator)
    - Proofs
    - Dependent types
    - Effects
    - Use Zarf as an IR

Questions
(and maybe answers)