A Dependently Typed Assembly Language

Hongwei Xi
University of Cincinnati
hwxi@ececs.uc.edu

Robert Harper
Carnegie Mellon University
rwh@cs.cmu.edu

ABSTRACT

We present a dependently typed assembly language (DTAL) in which the type system supports the use of a restricted form of dependent types, reaping some benefits of dependent types at the assembly level. DTAL improves upon TAL, enabling certain important compiler optimizations such as run-time array bound check elimination and tag check elimination. Also, DTAL formally addresses the issue of representing sum types at assembly level, making it suitable for handling not only datatypes in ML but also dependent datatypes in Dependent ML (DML).

1. INTRODUCTION

A compiler for a realistic programming language is often large and complex. Though it is highly desirable to establish the correctness of such a compiler, there seems no effective approach to reaching this goal currently. Instead, the ongoing research on certifying compilers attempts to partially address this problem from a different angle.

Suppose we have a compiler that translates source program e into target code $|e|$; if e possesses some property P (e.g. e is terminating) that we know $|e|$ must also possess if the compiler is implemented correctly, we can then design the compiler to produce a verifiable certificate asserting that $|e|$ possesses the property P; if the certificate is successfully verified, our confidence in the compiler is raised; otherwise, a compiler error needs to be located and then fixed.

In DML [18, 13], a functional programming language that supports the use of a restricted form of dependent types, a well-typed program is both type safe (which excludes, for examples, programs that attempt to add an integer to a floating point number) and memory safe (which excludes stray memory accesses). If we compile a well-typed program in DML into some target code at assembly level, the target code should also be both type safe and memory safe. Obviously, the immediate question is how both type safety and memory safety can be captured at assembly level. In this paper, we address this question by designing a dependently typed assembly language in which the dependent types can capture both type safety and memory safety.

ICFP'01, September 3-5, 2001, Florence, Italy.

Specific approaches to certification include proof-carrying code (PCC) (adopted in Touchstone [8]) and type systems (adopted in TIL [11]). In PCC, both type safety and memory safety are expressed by (first-order) logic assertions about program variables and are checked by a verification condition generator and a theorem prover, and code is then certified by an explicit representation of the proof. In TIL, type safety is expressed by type annotations and is checked by a type checker and no additional certification is required. The Touchstone approach draws on established results for verification of first-order imperative programs. The TIL approach draws on established methods for designing and implementing type systems, making it unclear (*a priori*) that it can be extended to low-level languages or to account for memory safety.

Typed Assembly Language [7] is introduced by Morrisett et al., where a form of type system is designed at assembly-level suitable for compiling functional languages and a compilation from System F to TAL is given. TAL provides both type safety and memory safety, but at the cost of making critical instructions such as array subscripting atomic to ensure memory safety. For instance, each array subscripting instruction in TAL involves checking whether a given array index is between the lower and upper bounds of the array before fetching the data item.

We enrich TAL to allow for more fine-grained control over memory safety so as to support array bound check elimination, hoisting bound checks out of loops, efficient representation of sum types, etc. We draw on the formalism of dependent types to extend TAL with such a concept. However, we cannot rely directly on standard systems of dependent types [4] for languages with computational effects. For instance, it is entirely unclear what it means to say that A is an array of length x for some mutable variable x: if we update x with a different value, this changes the type of A but A itself is unchanged! Drawing on our experience with a restricted form of dependent types in DML [18], we introduce a clear separation between ordinary run-time expressions and a distinguished family of index expressions, linked by singleton types of form $\mathrm{int}(x)$: every integer expression of type $\mathrm{int}(x)$ must have value equal to x. The index expressions are chosen from an integer domain in this paper. Given an expression e (in DML), checking whether e has type $\mathrm{int}(x)$ (written as $e : \mathrm{int}(x)$) involves non-trivial equational reasoning about the run-time behavior of e. For instance, $e : \mathrm{int}(3)$ means that e, when evaluated, must eval-

```
{m:nat, n:nat | m <= n}
void copy(int src[m], int dst[n]) {
  var: int i, length;;
  length = arraysize(src);
  for (i = 0; i < length; i = i + 1) {
    dst[i] = src[i];
  }
  return;
}
```

Figure 1: A copy function in Xanadu

uate to 3. Clearly, $3 : \mathrm{int}(3)$, and perhaps, $1+2 : \mathrm{int}(3)$, but it is, in general, undecidable whether an arbitrary (possibly effectful) e has type int(3). This is where theorem proving and constraint satisfaction comes into the picture.

It is difficult to read assembly code. In the following presentation, we will occasionally use programs in Xanadu [15], a dependently typed imperatively programming language with C-like syntax, to facilitate the presentation of DTAL. We could also use programs in DML for this purpose but the great difference between DML and DTAL would make this alternative less desirable. The Xanadu program in Figure 1 implements a copy function on arrays. The function header in the program states that for all natural numbers m and n satisfying $m \leq n$ the function takes two integer arrays of sizes m and n, respectively, and returns no value. Note that `{m:nat, n:nat | m <= n}` is a universal quantifier and

`int src[m]` and `int dst[n]`

mean that `src` and `dst` are integer arrays of sizes m and n, respectively. We use `var:` to start variable declaration, which ends with `;;`. Furthermore, the function `arraysize` returns the size of an array. Note that the type index `m` is not available at run-time and we use `arraysize` here to get an integer equal to `m` (or literally, an integer of type int(m)).

The DTAL code in Figure 2 corresponds to the Xanadu program. Note that `r1`, ..., `r5` are registers. The instruction `arraysize r3, r1` is non-standard, which means that we store into `r3` the size of the array to which `r1` points. The branch instruction `bgte r5, finish` jumps to the label `finish` if the integer in `r5` is greater than or equal to zero. Also `load r5, r1(r4)` means that we store into `r5` the content of the ith cell in the array to which `r1` points, where i is the integer stored in `r4`. The store instruction is interpreted similarly.

Every label in the code is associated with a *dependent type*. The dependent type associated with the label `loop` basically means that there exist a natural number `m` and a natural number `n` satisfying `m` $\leq$ `n` and a natural number `i` such that `r1, r2, r3, r4` are of types `int array(m), int array(n), int(m), int(i)`, respectively, that is, they are an integer array of size `m`, an integer array of size `n`, an integer of value `m` and an integer of value `i`. This enables us to state, for instance, that the type of `r1` depends on the value in `r3`. The type system of DTAL guarantees that these properties are satisfied when the code execution reaches the label `loop`.

The DTAL code is well-typed, which guarantees that the integer in `r4` is always a natural number and its value is always less than the size of the array to which `r1` (`r2`) points when the load (store) instruction is executed. [1] In other words, it can be statically verified that there is no need for run-time array bound checking in this case. Although this is a very simple example, it is nonetheless impossible to infer that the store instruction is safe without the dependent type associated with the label `loop`. In DTAL, array access is separated from array bound checks and the type system of DTAL guarantees that the execution of well-typed DTAL can never perform out-of-bounds array access. It is this separation that makes array bound check elimination possible. In the case where it is impossible to prove in the type system of DTAL whether an array access may be out-of-bounds, run-time array bound checks can be inserted to ensure safety.

We also address in DTAL the issue of representing sum types at assembly level. Furthermore, we demonstrate how dependent datatypes in DML can be translated into DTAL, allowing, for instance, an implementation of the list reverse function in DTAL that uses the type system of DTAL to guarantee this function to be length-preserving.

In a realistic setting, machine-level arithmetic is often modulo a power of 2, say, 2^{32}. This can be readily handled in our framework. For instance, we can assign the following type to $+$ for handling (unsigned) addition modulo 2^{32}, where int_{32} is the sort $\{a : int \mid 0 \leq a < 2^{32}\}$.

$$\Pi i : int_{32}.\Pi j : int_{32}.\mathrm{int}(i) * \mathrm{int}(j) \to \mathrm{int}((i+j) \mod 2^{32})$$

The reason that we do not treat modulo arithmetic in this paper is merely for a less involved presentation.

The main contribution of the paper is a formulation of a dependent type system for a language at assembly level that (a) is non-trivial for reasons outlined previously, (b) generalizes TAL to allow for capturing significant loop-based optimizations, (c) yields an application of dependent types to managing low-level representation of sum types, setting up some machinery needed for compiling dependent datatypes supported in DML into assembly level, and (d) provides an approach to certification based on type-checking. One tradeoff is that we presume that the constraint solver is part of trusted computing base in order for the recipient to verify the code it receives. Future work might include some means of formally representing proofs of constraints so that the constraint solver can be moved out of the trusted computing base.

Also, it is to be studied what are the advantages and disadvantages of using a DTAL-like language as the target language of a compiler. When compared with the work in DML and Xanadu, novelties in DTAL include:

- Datatype representation at assembly level. For instance, assume that a function in DML is given the type $\Pi a : nat.(\alpha)list(n) \to (\alpha)list(n)$, that is, it is length preserving; how can such a property be translated into low-level code?

- Control flow at assembly level that involves dependent types. There are simply no jumps, conditional or unconditional, in either DML or Xanadu, but we have to deal with such language features in DTAL.

In general, the design of DML and Xanadu is more concerned with type inference while the design of DTAL is more concerned with type checking as the types in DTAL are to

[1] This point should become clear if one reasons about instruction 4 and 5 in the code.

```
copy:   {m:nat, n:nat | m <= n} [r1: int array(m), r2: int array(n)]
        arraysize r3, r1        // obtain the size of source array
        mov       r4, 0         // initialize the loop count to 0

loop:   {m:nat, n:nat | m <= n, i:nat}
        [r1: int array(m), r2: int array(n), r3: int(m), r4: int(i)]
        sub       r5, r4, r3    // r5 <- r4 - r3
        bgte      r5, finish    // r4 >= r3
        load      r5, r1(r4)    // safe load
        store     r2(r4), r5    // safe store
        add       r4, r4, 1     // increase the count by 1
        jmp       loop          // loop again

finish: []
        halt // it can also return to the caller if needed
```

Figure 2: A copy function implemented in DTAL

be generated by a compiler. For instance, some of the typing rules in DTAL are *not* syntax directed, and annotations may need to be generated by a compiler in DTAL code to direct type-checking. We consider this to be a crucial point in the design of DTAL.

We organize the paper as follows. The syntax of DTAL is given in Section 2. We then form evaluation and typing rules so as to assign dynamic and static semantics to DTAL, respectively. We, however, postpone until Section 3 the treatment of constraints, which are generated during type-checking programs in DTAL. In Section 4, we give a detailed example explaining how type-checking is performed in DTAL. The soundness of the type system of DTAL is stated in Section 5 and an extension of DTAL to handle sum types is given in Section 6. We then in Section 7 mention a type-checker for DTAL and a compiler which compiles Xanadu, a language resembling Safe C [9] and Popcorn [6] with C-like syntax, into DTAL. The rest of the paper discusses some closely related work and future directions.

2. DTAL

In this section we present a dependently typed assembly language (DTAL), forming both dynamic and static semantics for DTAL.

2.1 Syntax

We assume that there are a fixed number n_r of registers. A register file R is a finite mapping from the set $\{0, 1, \ldots, n_r - 1\}$ into types. The intent is to capture some type information on registers with R. The syntax for DTAL is given in Figure 3. Note that stacks, which are treated in [16], are omitted here for simplicity, though we do use stacks in some code example. One may simply think of a stack as an infinite list of registers. Also, we omit tuples, which can be handled as in TAL.

Intuitively speaking, dependent types are types which depend on the values of language expressions. For instance, we may form a type (int)array(x) to mean that every heap pointer of this type points to an integer array of size x, where x is the expression on which this type depends. We use the name *type index expression* for such an expression. We restrict type index expressions to an integer domain. The justification for this choice is that we have used this domain to eliminate array bound checks effectively [17].

We present the syntax for type index expressions in Figure 4, where we use a to range over type index variables and i for fixed integers. Note that the language for type index expressions is typed. We use *sorts* for the types in this language in order to avoid potential confusion. We use $\cdot$ for the empty index context and omit the standard sorting rules for this language. The subset sort $\{a : \gamma \mid P\}$ stands for the sort for those elements of sort γ satisfying the proposition P. For example, we use *nat* as an abbreviation for $\{a : int \mid a \geq 0\}$.

We postpone the treatment of constraint satisfaction in this type index language until Section 3 for simplicity of exposition. However, we informally explain the need for constraints through the DTAL code in Figure 2. Notice that register `r4` is assumed to be of type int(i_1) for some natural number i_1 when the execution reaches the label `loop`. The type of `r4` changes into int($i_1 + 1$) after the execution of the instruction `add r4, r4, 1`. Then the execution jumps back to the label `loop`. This jump requires it to be verified (among many other requirements) that `r4` is of type int(i_2) for some natural number i_2. Therefore, we need to prove that $i_1 + 1$ is a natural number under the condition that i_1 is a natural number. This is a constraint, though it is trivial in this case. In general, type-checking in DTAL involves solving a great number of constraints of this form.

We use *top* for the type of uninitialized registers and assume that a register is initialized if it is not of type *top*. A block $B = \lambda\Delta\lambda\phi.(R, I)$ roughly means that B is polymorphic in type variable context Δ and index variable context ϕ. We may omit $\lambda\Delta$ ($\lambda\phi$) if Δ (ϕ) is empty. In order to execute the block on an abstract machine, we need to find substitutions Θ and θ for Δ and ϕ, respectively, such that the current machine state entails the state $R[\Theta][\theta]$ and then execute $I[\Theta][\theta]$. The entailment of R means that the type assignment to registers in R correctly reflects the types of registers in the current abstract machine. For instance, if R indicates that an integer is in a register r, then an integer must be stored in r in the abstract machine. A state type *state*$(\lambda\Delta.\lambda\phi.R)$, when associated with a label, means that there are substitutions Θ and θ for Δ and ϕ, respectively, such that the current abstract machine state entails $R[\Theta][\theta]$ whenever the execution reaches the label. The explanation here assumes that we carry types around when we evaluate DTAL code. Of course, we do not actually need to carry types around in practice when we evaluate DTAL code as it

type variables	α		
state types	σ	::=	$state(\lambda\Delta.\lambda\phi.R)$
regfile types	R	::=	$[r_0 : \tau_0, \ldots, r_{n_r-1} : \tau_{n_r-1}]$
types	τ	::=	$\alpha \mid \sigma \mid top \mid unit \mid \mathrm{int}(x) \mid \tau\ \mathrm{array}(x) \mid \exists\phi.\tau$
type erasures	ϵ	::=	$\alpha \mid top \mid unit \mid \mathrm{int} \mid \epsilon\ \mathrm{array}$
type variable contexts	Δ	::=	$\cdot_{\mathrm{tv}} \mid \Delta, \alpha$
registers	r	::=	$r_0, \ldots, r_{n_r-1}$
instructions	ins	::=	$aop\ r_d, r_s, v \mid bop\ r, v \mid$ $\mathtt{arraysize}\ r_d, r_s \mid$ $\mathtt{mov}\ r, v \mid \mathtt{load}\ r_d, r_s(v) \mid \mathtt{store}\ r_d(v), v_s \mid$ $\mathtt{newarray}[\tau]\ r, r', r'' \mid \mathtt{jmp}\ v \mid \mathtt{halt}$
fixed integers	i	::=	$\cdots \mid -1 \mid 0 \mid 1 \mid \cdots$
constants	c	::=	$\langle\rangle \mid i \mid l$
values	v	::=	$c \mid r$
instruction sequences	I	::=	$\mathtt{jmp}\ v \mid \mathtt{halt} \mid ins; I$
blocks	B	::=	$\lambda\Delta.\lambda\phi.(R, I)$
arithmetic ops	aop	::=	$\mathtt{add} \mid \mathtt{sub} \mid \mathtt{mul} \mid \mathtt{div}$
branch ops	bop	::=	$\mathtt{beq} \mid \mathtt{bne} \mid \mathtt{blt} \mid \mathtt{blte} \mid \mathtt{bgt} \mid \mathtt{bgte}$
labels	l		
label mappings	Λ	::=	$\{l_1 : \sigma_1, \ldots, l_n : \sigma_n\}$
programs	P	::=	$l_1 : B_1; \ldots ; l_n : B_n$

Figure 3: Syntax for DTAL

index variables	a		
index expressions	x, y	::=	$a \mid i \mid x + y \mid x - y \mid x * y \mid x \div y$
index propositions	P	::=	$x < y \mid x \le y \mid x = y \mid x \ge y \mid x > y \mid \neg P \mid P_1 \wedge P_2 \mid P_1 \vee P_2$
index sorts	γ	::=	$int \mid \{a : \gamma \mid P\}$
index contexts	ϕ	::=	$\cdot \mid \phi, a : \gamma \mid \phi, P$

Figure 4: Syntax for type index expressions

$$
\begin{array}{l}
P = (\mathtt{copy} : B_1, \mathtt{loop} : B_2, \mathtt{finish} : B_3) \\
\Lambda(P) = \{\mathtt{copy} : \sigma_1, \mathtt{loop} : \sigma_2, \mathtt{finish} : \sigma_3\} \\
J(P) = \mathtt{copy}; I_1; \mathtt{loop}; I_2; \mathtt{finish}; \mathtt{halt} \\
B_1 = \lambda(m : nat, n : nat, m \le n).(R_1, I_1) \\
B_2 = \lambda(m : nat, n : nat, m \le n, i : nat).(R_2, I_2) \\
B_3 = (R_{empty}, \mathtt{halt}) \\
\sigma_1 = state(\lambda(m : nat, n : nat, m \le n).R_1) \\
\sigma_2 = state(\lambda(m : nat, n : nat, m \le n, i : nat).R_2) \\
\sigma_3 = state(R_{empty})
\end{array}
$$

Figure 5: The representation of the program in Fig 2

is clear types play no rôle in evaluation of DTAL code. This is precisely like the case where a well-typed ML program is evaluated.

We use J for a general instruction sequence in the following presentation, which consists of a sequence of instructions or labels. Given a block $B = \lambda\Delta.\lambda\phi.(R, I)$, we write $\sigma(B)$ for $state(\lambda\Delta.\lambda\phi.R)$ and $I(B)$ for I. Also we define functions Λ and J on program $P = l_1 : B_1; \ldots ; l_n : B_n$ as follows.

$$
\begin{array}{rcl}
\Lambda(P) & = & \{l_1 : \sigma(B_1), \ldots, l_n : \sigma(B_n)\} \\
J(P) & = & l_1; I(B_1); \ldots ; l_n; I(B_n)
\end{array}
$$

We refer $\Lambda(P)$ as the label mapping of P, in which we require that all labels be distinct. For a valid program P, all labels in $J(P)$ must be declared in $\Lambda(P)$. In all the examples of DTAL code that we present in this paper, we attach the state type σ of a label l to the label explicitly in the program, and the label mapping of the program can be immediately extracted from the code if necessary. We explain these definitions in Figure 5, where the program P is given in Figure 2; I_1 and I_2 are the sequences of instructions between the labels `copy` and `loop` and those between labels `loop` and `finish`, respectively. R_1 is a mapping which maps 1 and 2 to (int)array(m) and (int)array(n), respectively, and $R_1(i) = top$ for $i \neq 1, 2$; R_2 maps $1, 2, 3$ and 4 to (int)array(m), (int)array(n), int(m) and int(i), respectively, and $R_2(i) = top$ for $i \neq 1, 2, 3, 4$; $R_{empty}(i) = top$ for i in all its domain. Note that we write int for $\exists a : int.\mathrm{int}(a)$, that is, int is the sum of all singleton types int(a), where a ranges over integers.

The following erasure function $\|\cdot\|$ transforms types into type erasures, that is, non-dependent types.

$$
\begin{array}{llll}
\|top\| = top & \|unit\| = unit & \|\alpha\| = \alpha & \|\mathrm{int}(x)\| = \mathrm{int} \\
\|\sigma\| = unit & \|\tau\ \mathrm{array}(x)\| = \|\tau\|\ \mathrm{array} & & \|\exists\phi.\tau\| = \|\tau\|
\end{array}
$$

It can be readily verified after the presentation of DTAL that DTAL becomes a TAL-like language if one erases all syntax related to type index expressions. In this TAL-like language, the erasure of a program is well-typed if it is well-typed in DTAL. In this respect, DTAL generalizes TAL. We stress the erasure property because it indicates that DTAL does not make more programs typable than TAL but, instead, can assign more accurate types to programs.

2.2 Dynamic Semantics

$$\frac{\|\tau\| = (\epsilon)\text{array} \quad J(ic) = \texttt{newarray}[\tau]\ r, r', r'' \quad \mathcal{M}(r') = n \geq 0 \quad h \notin \mathbf{dom}(\mathcal{H})}{(ic, \mathcal{M}) \to_P (ic+1, \mathcal{M}[h \mapsto (\mathcal{M}(r''), \ldots, \mathcal{M}(r''))][r \mapsto h])}\ \textbf{(eval-newarray)}$$

$$\frac{J(ic) = \texttt{load}\ r_d, r_s(v) \quad \mathcal{H}(\mathcal{M}(r_s)) = (hc_0, \ldots, hc_{n-1}) \quad \mathcal{M}(v) = i \quad 0 \leq i < n}{(ic, \mathcal{M}) \to_P (ic+1, \mathcal{M}[r_d \mapsto hc_i])}\ \textbf{(eval-load)}$$

$$\frac{\begin{array}{c} J(ic) = \texttt{store}\ r_d(v), v_s \quad \mathcal{M}(r_d) \mapsto h \quad \mathcal{M} = (\mathcal{H}, \mathcal{R}) \\ \mathcal{H}(h) = (hc_0, \ldots, hc_{n-1}) \quad \mathcal{M}(v) = i \quad 0 \leq i < n \quad \mathcal{M}(v_s) = hc \end{array}}{(ic, \mathcal{M}) \to_P (ic+1, (\mathcal{H}[h \mapsto (hc_0 \ldots, hc_{i-1}, hc, hc_{i+1}, \ldots, hc_{n-1})], \mathcal{R}))}\ \textbf{(eval-store)}$$

$$\frac{J(ic) = \texttt{halt}}{(ic, \mathcal{M}) \to_P \texttt{HALT}}\ \textbf{(eval-halt)}$$

Figure 6: Some evaluation rules for DTAL

We use an abstract machine for assigning operational semantics to DTAL, which is a standard approach. A machine state $\mathcal{M}$ is a pair $(\mathcal{H}, \mathcal{R})$, where $\mathcal{H}$ and $\mathcal{R}$ are finite mappings which stand for heap and register file, respectively.

The domain $\mathbf{dom}(\mathcal{H})$ of $\mathcal{H}$ is a set of heap addresses, the domain $\mathbf{dom}(\mathcal{R})$ of $\mathcal{R}$ is $\{0, \ldots, n_r - 1\}$. We do not specify how a heap address is represented, but the reader can simply assume it to be a natural number. Given $h \in \mathbf{dom}(\mathcal{H})$, $\mathcal{H}(h)$ is a tuple $(hc_0, \ldots, hc_{n-1})$ such that for $i = 0, \ldots, n-1$, every hc_i is either a heap address or a constant. Given $i \in \mathbf{dom}(\mathcal{R})$, $\mathcal{R}(i)$ is either a heap address or a constant.

Given a program P, $\Lambda = \Lambda(P)$ associates every label in $J = J(P)$ with a state type σ. We use $length(J)$ for the length of the sequence J, counting both instructions and labels. We use $J(i)$ for the ith item in J, which is either an instruction or a label. Also we write $J^{-1}(l)$ for i if l is $J(i)$. This is well-defined since all labels in a program are distinct. We define a P-snapshot Q as either `HALT` or a pair $(ic, \mathcal{M})$ such that $0 \leq ic < length(J)$. The relation $(ic, \mathcal{M}) \to_P (ic', \mathcal{M}')$ means that the current machine state $\mathcal{M}$ transforms into $\mathcal{M}'$ after executing the instruction $J(ic)$ and the instruction counter is set to ic'.

Given $\mathcal{M} = (\mathcal{H}, \mathcal{R})$, we define the following.

$$\mathcal{M}(v) = \begin{cases} \langle\rangle & \text{if } v \text{ is } \langle\rangle; \\ i & \text{if } v \text{ is integer } i; \\ l & \text{if } v \text{ is label } l; \\ \mathcal{R}(i) & \text{if } v \text{ is the } i\text{th register } r_i. \end{cases}$$

Given a finite mapping f and an element x in the domain of f, we use $f(x)$ for the value to which f maps x, and $f[x \mapsto v]$ for the mapping such that

$$f[x \mapsto v](y) = \begin{cases} f(y) & \text{if } y \text{ is not } x; \\ v & \text{if } y \text{ is } x. \end{cases}$$

Clearly, $f[x \mapsto v]$ is also meaningful when x is not already in the domain of f. In this case, we simply extend the domain of f with x.

We use the notation $\mathcal{R}[r \mapsto hc]$ to mean that we update the content of register r with hc, that is, $\mathcal{R}[r \mapsto hc]$ is really $\mathcal{R}[i \mapsto hc]$, where i is the numbering of register r. Also we use $\mathcal{M}[r \mapsto hc]$ for $(\mathcal{H}, \mathcal{R}[r \mapsto hc])$ given $\mathcal{M} = (\mathcal{H}, \mathcal{R})$.

We present some evaluation rules for DTAL in Figure 6. We do not consider garbage collection in this abstract machine, and therefore the typing of the heap can only be affected by the memory allocation instructions `newarray`. Notice that the rules **(eval-load)** and **(eval-store)** imply that an out-of-bounds array access stalls the abstract machine. These rules also indicate that the length of the tuple $\mathcal{H}(h)$ can always be determined for every $h \in \mathbf{dom}(\mathcal{H})$ at run-time. We will soon design a type system for DTAL and prove that $0 \leq i < n$ in both rules **(eval-load)** and **(eval-store)** always holds when these rules are applied during the evaluation of a well-typed DTAL program. Therefore, there is no need for determining the length of the tuple $\mathcal{H}(h)$ for every $h \in \mathbf{dom}(\mathcal{H})$ if we only evaluate well-typed DTAL programs. In the case where it cannot be determined in the type system of DTAL whether a subscript is within the bounds of an array, the array subscripting instruction is ill-typed and thus rejected. This sounds like a severe restriction, but it is not because we can always insert run-time array bound checks to make the instruction typable in DTAL (we give such an example at the end of Section 2.3).

The rule **(eval-newarray)** is non-standard. If $\|\tau\|$ is of form (ϵ)array, then $\texttt{newarray}[\tau]r, r', r''$ allocates n new word-sized memory on heap, where n is the integer stored in r', and initializes each word with the content in r'' and then stores a point in r which points to the allocated memory. We emphasize that h must be new in the rule **(eval-newarray)**, that is, h is not already in the domain of $\mathcal{H}$. The typing consequences of this memory allocation instructions is explained in the next section, where the typing rule **(type-newarray)** is introduced.

Let us call a program well-structured if its evaluation halts normally (when the rule **(eval-halt)** is applied) or continues forever. In other words, the evaluation of a well-structured program can never be stuck. Certainly it is undecidable to determine precisely whether a program is well-structured, but this is also less relevant. We intend to find a conservative approach to examining whether a program is well-structured. Such an approach must be sound, that is, it can only accept well-structured programs. For instance, a straightforward approach is to adopt a method based on TAL for type-safety and then insert run-time checks for all array operations. Unfortunately, this approach seems too conservative, making it impossible to eliminate array bound checks. Notice that this is essentially the case in all JVML verifiers. In the next section, we present a less conservative approach based on a dependent type system.

2.3 Static Semantics

We present the typing rules for DTAL in this section. Note that we use an array representation for a register file R. We

$$\frac{}{\phi;\Delta;R \vdash_\Lambda \langle\rangle : unit}\ \textbf{(type-unit)}$$

$$\frac{}{\phi;\Delta;R \vdash_\Lambda i : \mathrm{int}(i)}\ \textbf{(type-int)}$$

$$\frac{\Lambda(l)=\sigma}{\phi;\Delta;R \vdash_\Lambda l : \sigma}\ \textbf{(type-label)}$$

$$\frac{0 \le i < n_r}{\phi;\Delta;R \vdash_\Lambda r_i : R(i)}\ \textbf{(type-reg)}$$

$$\frac{\phi;\Delta;R \vdash_\Lambda v : \tau_1 \quad \phi;\Delta \models \tau_1 \le \tau_2}{\phi;\Delta;R \vdash_\Lambda v : \tau_2}\ \textbf{(type-sub)}$$

Figure 7: Typing rules for integers, labels, registers

omit the standard rules for forming legal types and assume that all types are well-formed in the following presentation.

We use a judgment of form $\phi;\Delta;R \vdash_\Lambda v : \tau$ to mean that value v is assigned type τ under the context $\phi;\Delta;R$ and the label mapping Λ. The label mapping Λ is always fixed when we type-check a program, and therefore we will omit it if this causes no confusion. The rules in Figure 7 are for typing unit, integers, labels and registers.

We present some typing rules for DTAL in Figure 8. We use θ and Θ for index and type variable substitutions, respectively, which are defined as usual. Given a term $\bullet$ such as a type or a register file, we write $\bullet[\Theta]$ ($\bullet[\theta]$) for the result of applying Θ (θ) to $\bullet$. A judgment of form $\phi;\Delta;R \vdash I$ means that the instruction sequence I is well-typed under context $\phi;\Delta;R$. The notation $R[r:\tau]$ means that we update the type of register r to τ in R, that is, if r is the ith register, then we update the value of $R(i)$ with τ. We use the rule **(type-newarray)** for typing arrays allocated on heap. We have explained in the previous section how memory allocation is performed. Also we require that the index variables declared in ϕ' in the rule **(type-open-reg)** have no free occurrence in the conclusion of the rule.

The typing rule **(type-add)** indicates that the type of register r_d become $\mathrm{int}(x+y)$ after the instruction `add` r_d, r_s, v is executed, where we assume that r_s and v have types $\mathrm{int}(x)$ and $\mathrm{int}(y)$, respectively. If arithmetic overflow is to be considered, we may require the instruction to be followed by an instruction that traps overflow; if an overflow occurs, we jump to a subroutine to handle it; otherwise, we know r_d indeed has type $\mathrm{int}(x+y)$.

We give some explanation on the rule **(type-beq)**. We use $\phi \vdash \theta : \phi'$ to mean that θ is a substitution for ϕ' under ϕ, that is, for every $a:\gamma$ declared in ϕ', $\phi \vdash \theta(a) : \gamma$ is derivable and for every P in ϕ', $\phi \models P[\theta]$ is satisfiable. The explanation for $\phi;\Delta \vdash \Theta : \Delta'$ is similar. Suppose that we type-check `beq`$;r,v;I$ under $\phi;\Delta;R$; we first check that r has type $\mathrm{int}(x)$ for some x; we then type-check I under $\phi, x \ne 0;\Delta;R$ ($x \ne 0$ is added into ϕ since the jump is not taken in this case); we also verify that v has a state type and $\phi, x = 0;\Delta;R$ entails the state type ($x = 0$ is added to ϕ since the jump is taken in this case). The typing rules for other conditional jumps are similar.

We sketch a case where a DTAL program that does not type-check can be modified to type-check with the insertion of a run-time array bound check. Assume that we want to type-check `load` $r_d, r_s(v); I$ under $\phi;\Delta;R$, and we have verified that r_s and v have types τ array(x) and $\mathrm{int}(y)$, respectively, and we can prove $\phi \models 0 \le y$ but not $\phi \models y < x$; we can then insert the following (where `subscript` is the entry to some routine that handles errors) in front of the load instruction, and this insertion guarantees that $x-y>0$ is already added to ϕ when the load instruction is type-checked, making sure that $y < x$ is provable.

`arraysize` r, r_s; `sub` r, r, v; `blte` $r,$ `subscript`;

A dual case is to remove a redundant array bound check. For instance, we want to type-check `blt` $r,$ `subscript`$; I$ under $\phi;\Delta;R$; suppose that r has type $\mathrm{int}(x)$ for some x and $\phi \models x \ge 0$ can be proven; this implies that `blt` $r,$ `subscript` can never branch and thus this instruction can be removed.

We use $\vdash P$[well-typed] to mean that a program $P = (l_1 : B_1, \ldots, l_n : B_n)$ is well-typed and the following rule is for typing a program, where Λ is the label mapping of P.

$$\frac{\vdash_\Lambda B_1[\text{well-typed}] \quad \cdots \quad \vdash_\Lambda B_n[\text{well-typed}]}{\vdash P[\text{well-typed}]}$$

Given a block $B = \lambda\Delta.\lambda\phi.(R, I)$, the rule for deriving $\vdash_\Lambda B$[well-typed] is given as follows.

$$\frac{\phi;\Delta;R \vdash_\Lambda I}{\vdash_\Lambda B[\text{well-typed}]}\ \textbf{(type-block)}$$

3. TYPE EQUALITY AND COERCION

As we have mentioned before, a novelty in DML is the separation between language expressions and type index expressions. This notion of separation seems indispensable when we intend to form a dependent type system for an imperative language such as DTAL. For instance, it is completely unclear at the moment how a register can be used as a type index expression, since it is mutable. The separation allows us to simply avoid such a problematic issue. Another advantage is that the separation enables us to choose a relatively simple domain for type index expressions so that constraints (on type index expressions) generated during type-checking can be efficiently solved. This is crucial to the design of a practical type-checking algorithm. In this section, we present type equality and coercion, which lead to constraint generation in type-checking.

In the presence of dependent types, it is no longer trivial to check whether two types are equivalent. For instance, we have to prove that the constraint $1+1=2$ in order to claim $\mathrm{int}(1+1)$ is equivalent to $\mathrm{int}(2)$. In other words, type equality is modulo constraint satisfaction. Similarly, type coercion also involves constraint satisfaction.

We use Φ for index constraints,

$$\Phi \quad ::= \quad \top \mid P \mid P \supset \Phi \mid \forall a:\gamma.\Phi$$

and $\phi \models P$ for a satisfiability relation, stating that $(\phi)P$ is satisfiable in the domain of integers, where $(\phi)P$ is defined below.

$$(\cdot)\Phi = \Phi \qquad (\phi, a:int)\Phi = (\phi)\forall a:int.\Phi$$
$$(\phi, a:\{a:\gamma \mid P\})\Phi = (\phi, a:\gamma)(P \supset \Phi)$$
$$(\phi, P)\Phi = (\phi)(P \supset \Phi)$$

For instance, the satisfiability relation $a:nat, b:int, a+1=b \models b>0$ holds since the following formula is true in the integer domain.

$$\forall a:int.a \ge 0 \supset \forall b:int.a+1=b \supset b>0$$

$$\frac{\phi, \phi'; \Delta; R[r : \tau] \vdash I}{\phi; \Delta; R[r : \exists \phi'.\tau] \vdash I} \textbf{ (type-open-reg)}$$

$$\frac{\phi; \Delta; R \vdash r' : int(x) \quad \phi \models x \geq 0 \quad \phi; \Delta; R \vdash r'' : \tau \quad \phi; \Delta; (R[r : \tau \text{ array}(x)]) \vdash I}{\phi; \Delta; R \vdash \texttt{newarray}[\tau]\ r, r', r''; I} \textbf{ (type-newarray)}$$

$$\frac{\phi; \Delta; R \vdash r_s : \text{int}(x) \quad \phi; \Delta; R \vdash v : \text{int}(y) \quad \phi; \Delta; (R[r_d : \text{int}(x+y)]) \vdash I}{\phi; \Delta; R \vdash \texttt{add}\ r_d, r_s, v; I} \textbf{ (type-add)}$$

$$\frac{\phi; \Delta; R \vdash r_s : \tau \text{ array}(x) \quad \phi; \Delta; R \vdash v : \text{int}(y) \quad \phi \models 0 \leq y < x \quad \phi; \Delta; R[r_d : \tau] \vdash I}{\phi; \Delta; R \vdash \texttt{load}\ r_d, r_s(v); I} \textbf{ (type-load-array)}$$

$$\frac{\phi; \Delta; R \vdash r_d : \tau \text{ array}(x) \quad \phi; \Delta; R \vdash v : \text{int}(y) \quad \phi \models 0 \leq y < x \qquad \phi; \Delta; R \vdash v_s : \tau \quad \phi; \Delta; R \vdash I}{\phi; \Delta; R \vdash \texttt{store}\ r_d(v), v_s; I} \textbf{ (type-store-array)}$$

$$\frac{\phi; \Delta; R \vdash v : state(\lambda\Delta'.\lambda\phi'.R') \quad \phi \vdash \theta : \phi' \quad \phi; \Delta \vdash \Theta : \Delta' \quad \phi; \Delta; R \models_c R'[\Theta][\theta]}{\phi; \Delta; R \vdash \texttt{jmp}\ v; I} \textbf{ (type-jmp)}$$

$$\frac{\begin{array}{c}\phi; \Delta; R \vdash r : \text{int}(x) \quad \phi, x \neq 0; \Delta; R \vdash I \quad \phi; \Delta; R \vdash v : state(\lambda\Delta'.\lambda\phi'.R') \\ \phi, x = 0 \vdash \theta : \phi' \quad \phi, x = 0; \Delta \vdash \Theta : \Delta' \quad \phi, x = 0; \Delta; R \models_c R'[\Theta][\theta]\end{array}}{\phi; \Delta; R \vdash \texttt{beq}\ r, v; I} \textbf{ (type-beq)}$$

Figure 8: The typing rules for DTAL

$$\frac{\phi; \Delta \vdash \tau : *}{\phi; \Delta \models \tau \leq top} \textbf{ (coerce-top)} \qquad \frac{\alpha \in \Delta}{\phi; \Delta \models \alpha \leq \alpha} \textbf{ (coerce-type-var)} \qquad \frac{\phi \models x = y}{\phi; \Delta \models \text{int}(x) \leq \text{int}(y)} \textbf{ (coerce-int)}$$

$$\frac{\phi, \phi'; \Delta \models \tau_1 \leq \tau_2}{\phi; \Delta \models \exists \phi'.\tau_1 \leq \tau_2} \textbf{ (coerce-exi-ivar-l)} \qquad \frac{\phi \vdash \theta : \phi' \quad \phi; \Delta \models \tau_1 \leq \tau_2[\theta]}{\phi; \Delta \models \tau_1 \leq \exists \phi'.\tau_2} \textbf{ (coerce-exi-ivar-r)}$$

$$\frac{\phi; \Delta \models \tau_1 \equiv \tau_2 \quad \phi \models x = y}{\phi; \Delta \models \tau_1 \text{ array}(x) \leq \tau_2 \text{ array}(y)} \textbf{ (coerce-array)} \qquad \frac{\phi; \Delta \models R(i) \leq R'(i) \quad \text{for } 0 \leq i < n_r}{\phi; \Delta; R \models_c R'} \textbf{ (coerce-reg)}$$

Figure 9: Some type coercion rules for DTAL

We currently only accept linear constraints, using linear integer programming to solve them. Though the constraint satisfaction is NP-complete, most constraints in practice are efficiently solved.

We write $\phi; \Delta \models \tau_1 \equiv \tau_2$ to mean that types τ_1 and τ_2 are equal under context $\phi; \Delta$. Similarly, we write $\phi; \Delta \models \tau_1 \leq \tau_2$ to mean that type τ_1 coerces into type τ_2 under context $\phi; \Delta$. Note that type coercion can simply be view as a form subtyping here. Some rules for type coercion are presented in Figure 9. Notice that for the rule **(coerce-exi-ivar-l)**, there is an obvious side condition requiring that the type τ_2 does not contain free occurrences of the index variables declared in ϕ'.

The rules for type equality are similar and thus omitted.

For instance, the following derivation shows that the type $\exists a : nat.\text{int}(a)$ coerces into the type $\exists b : int.\text{int}(b)$, where the the top applied rule is **(coerce-exi-ivar-r)** and the other is **(coerce-exi-ivar-l)**.

$$\frac{\dfrac{a : nat \models a : int \quad a : nat; \cdot \models \text{int}(a) \leq \text{int}(a)}{a : nat; \cdot \models \text{int}(a) \leq \exists b : int.\text{int}(b)}}{\cdot; \cdot \models \exists a : nat.\text{int}(a) \leq \exists b : int.\text{int}(b)}$$

We have so far finished the presentation of the type system of DTAL, which is rather involved. We will use a concrete example in the next section to provide some explanation on type-checking before proceeding to establish the soundness of the type system.

4. AN EXAMPLE

We demonstrate some key steps involved in type-checking the DTAL code in Figure 2. We stick to the notations given in Figure 5. Let ins_i be the ith instruction and $I_{2,i}$ be $ins_i; \ldots; ins_9$ for $4 \leq i \leq 9$. In order to derive $\vdash B_2$[well-typed], that is, to type block B_2, we need to derive the following.

$$m : nat, n : nat, m \leq n, i : nat; \cdot; R_2 \vdash I_2$$

Then there must be derivations $\mathcal{D}_i$ with a conclusion of form $\phi_i; \Delta_i; R_i \vdash I_{2,i}$ for $i = 4, \ldots, 9$. We list these contexts $\phi_i; \Delta_i; R_i$ in Figure 10. In the derivation of $\phi_6; \Delta_6; R_6 \vdash I_6$, the last rule is **(type-load-array)**, where we need to prove $\phi_6 \models 0 \leq i < m$. This is trivial since $i : nat$ and $i < m$ are assumed in ϕ_6. Similarly, we need to prove $\phi_7 \models 0 \leq i < n$ when deriving $\phi_7; \Delta_7; R_7 \vdash I_7$. This is also trivial since $m \leq n, i : nat, i < m$ are assumed in ϕ_7.

5. SOUNDNESS

By the type soundness of DTAL, we essentially mean that the evaluation of well-typed DTAL code either halts normally (when the instruction `halt` is executed) or goes on indefinitely. The main ingredient in the proof of the type

No.	ϕ	Δ	R
04	$m : nat, n : nat, m \leq n, i : nat$	$\cdot$	R_2
05	$m : nat, n : nat, m \leq n, i : nat$	$\cdot$	$R_2[r_5 : \text{int}(i - m)]$
06	$m : nat, n : nat, m \leq n, i : nat, i - m < 0$	$\cdot$	$R_2[r_5 : \text{int}(i - m)]$
07	$m : nat, n : nat, m \leq n, i : nat, i - m < 0$	$\cdot$	$R_2[r_5 : \text{int}]$
08	$m : nat, n : nat, m \leq n, i : nat, i - m < 0$	$\cdot$	$R_2[r_5 : \text{int}]$
09	$m : nat, n : nat, m \leq n, i : nat, i - m < 0$	$\cdot$	$R_2[r_5 : \text{int}][r_4 : \text{int}(i + 1)]$

Figure 10: Contexts $\phi_k; \Delta_k; R_k$ for $k = 4, \ldots, 9$

$$\frac{\phi, x = 0; \Delta \models \tau_0 \leq \tau \quad \cdots \quad \phi, x = n-1; \Delta \models \tau_{n-1} \leq \tau}{\phi; \Delta \models \mathit{choose}(x, \tau_0, \ldots, \tau_{n-1}) \leq \tau} \textbf{ (coerce-choose-l)}$$

$$\frac{\phi; \Delta \models \tau \leq \tau_i \quad \phi \models x = i}{\phi; \Delta \models \tau \leq \mathit{choose}(x, \tau_0, \ldots, \tau_{n-1})} \textbf{ (coerce-choose-r)}$$

Figure 11: Additional type coercion rules for sum types

soundness of DTAL is an entailment relation, for which we present a brief explanation.

Given a program P, we use J for the list consisting of labels and instructions in P and $J[ic]$ for the suffix of J starting with the ic^{th} item in J. Assume $\phi; \Delta; R \vdash J[ic]$ is derivable and there are substitutions θ and Θ for ϕ and Δ, respectively, such that $\mathcal{M} \models R[\Theta][\theta]$ holds, that is, $\mathcal{M}$ entails $R[\Theta][\theta]$. We use $\mathcal{H} \models hc : \tau$ to mean that hc has type τ under the heap mapping $\mathcal{H}$. For instance, we have $\mathcal{H} \models i : \text{int}(i)$. The following rule **(heap-array)** is for assigning array types.

$$\frac{\mathcal{H}(h) = (hc_0, \ldots, hc_{n-1}) \quad \mathcal{H} \models hc_0 : \tau \quad \cdots \quad \mathcal{H} \models hc_{n-1} : \tau}{\mathcal{H} \models h : (\tau)\text{array}(n)}$$

We write $(\mathcal{H}, \mathcal{R}) \models R$, that is, $(\mathcal{H}, \mathcal{R})$ entails R, if $\mathcal{H} \models \mathcal{R}(i) : R(i)$ holds for every $i \in \mathbf{dom}(R)$. In other word, $(\mathcal{H}, \mathcal{R}) \models R$ means that the content in each register does have the type assigned by R.

We state the type soundness theorem for DTAL below.

THEOREM 5.1. *Let $P = (l_1 : B_1; \ldots; l_n : B_n)$ be a program and $\Lambda = \Lambda(P)$. Assume $\vdash P[\text{well-typed}]$ is derivable and $\Lambda(l_1) = R_{empty}$, where R_{empty} maps each register to type top. For every machine state $\mathcal{M}_0$, If $(0, \mathcal{M}_0) \rightarrow_P^* (ic, \mathcal{M})$ then either $(ic, \mathcal{M}) \rightarrow_P$ HALT, or $(ic, \mathcal{M}) \rightarrow_P (ic', \mathcal{M}')$ for some ic' and $\mathcal{M}'$. In other words, the execution of a well-typed program in DTAL either halts normally or runs forever.*

The proof of this theorem is involved. We have to deal with a subtle issue involving shared pointers and impose some regularity condition on the heap mapping $\mathcal{H}$ in a machine state in order to establish the result. We give some brief explanation on this issue.

Suppose $\mathcal{H}(h) = (0)$ for some h, $\mathcal{R}(0) = \mathcal{R}(1) = h$, $R(0) = (\text{int})\text{array}(1)$ and $R(1) = (\text{nat})\text{array}(1)$, where we write nat for $\exists a : nat.int(a)$. We can now derive $(\mathcal{H}, \mathcal{R}) \models R$ since (0) can be viewed as both an integer array of size 1 and a natural number array of size 1. Clearly, if we store a negative integer into the array pointed by r_1, then the type of r_2 is invalidated because it no longer points to a natural number array.

Assume $\phi; \Delta; R \vdash J[ic]$ is derivable, $\mathcal{M}$ entails $R[\Theta][\theta]$ for some Θ and θ and $(\mathcal{M}, ic) \rightarrow_P (\mathcal{M}', ic')$, what we essentially need to prove is that $\phi'; \Delta'; R' \vdash J[ic']$ is derivable for some ϕ' and Δ' such that $\mathcal{M}'$ entails $R'[\Theta'][\theta']$. Unfortunately, the above example shows that this is not provable as it is simply false. In order to overcome the problem, we impose a regularity condition on the derivation of $\mathcal{M} \models R$. Roughly speaking, we associate type τ with heap address h whenever the rule **(heap-array)** is applied and a derivation is regular if a heap address is associated with at most one type. This notion of regularity is essentially the same as the notion of store typing in [2], which was used to address the circularity of references in ML. Clearly, there is no regular derivation for the above example: in order to derive $\mathcal{M} \models R$, we have to associate h with at least two distinct types int (when we derive $\mathcal{H} \models \mathcal{R}(0) : R(0)$) and nat (when we derive $\mathcal{H} \models \mathcal{R}(1) : R(1)$).

In essence, by a regular derivation of $(\mathcal{H}, \mathcal{R}) \models R$, we mean that there is a heap typing that maps each heap address $h \in \mathbf{dom}(H)$ to a fixed type and under this typing $\mathcal{R}(i)$ can be assigned the type $R(i)$, that is, the value in each register has the type that is delcared for the register. As a heap typing can never be altered (but it may be extended by the execution of `newarray`), We can then prove that if $\mathcal{M} \models R[\Theta][\theta]$ has a regular derivation then $\mathcal{M}' \models R'[\Theta'][\theta']$ also has a regular derivation, where we use the notation in the above paragraph. The proof bears a great deal of similarity to the soundness proof in [2].

In summary, if we start with an entailment that has a regular derivation, then all entailments in the proof of the type soundness of DTAL have regular derivations. Therefore, the scenario of shared pointers mentioned previously can never occur. This allows us to establish Theorem 5.1. Note the issue here, which we think is rather subtle to recognize, does not occur in either DML or TAL. Please see [16] for details.

6. EXTENSION WITH SUM TYPES

The programmer can declare in Xanadu a polymorphic union type as in Figure 12 for representing lists and then implement the length function. The concrete syntax `<'a> list` is for the type of lists in which all elements are of type `'a`

```
('a) union list with nat =
 {Nil(0); {n:nat} Cons(n+1) of 'a * <'a>list(n)}

('a){n:nat} int(n) length (xs: <'a> list(n)) {
  var: int x = 0;;
  invariant:
  [i:nat,j:nat | i+j=n] (xs:<'a>list(i), x:int(j))
  while (true) {
    switch(xs) {
      case Nil: return x;
      case Cons(_, xs): x = x + 1;
    }
  }
  exit; /* can never be reached */
}
```

Figure 12: A list length function in Xanadu

(we use `'a` for a type variable). Note that the union types in Xanadu correspond to datatypes in ML and the values of union types are decomposed through pattern matching. We informally explain the meaning of the `switch` statement in Figure 12; if `xs` matches the pattern `Nil`, the value of `x` is returned; if `xs` matches the pattern `Cons(_, xs)` (`_` is a wildcard), then we update `xs` with its tail and increase `x` by 1. The type following the keyword `invariant` states an invariant at the program point: `xs` is a list of length i and `x` is an integer of value j for some integers i, j satisfying $i + j = n$, where n is the length of the function argument.

A union type is internally represented as a sum type. In the case above, a tag is used to indicate whether the outmost constructor of a list is `Nil` or `Cons`. We can compile the length function essentially in the following manner; we initialize x with 0 and start the following loop; given a list xs, we perform a tag check to see whether it is `Nil`; if it is, we return x; otherwise, we know that the outmost constructor of xs must be `Cons` and it is unnecessary to perform another tag check; we can simply update xs with its tail, increase x by 1 and loop again.

We now extend the system of DTAL to handle sum types. In an implementation, we can use a pair on heap to represent a sum type $sum(\tau_0, \dots, \tau_{n-1})$, which is often written as $\tau_0 + \cdots + \tau_{n-1}$ in the literature. The first element of the pair is an integer i such that $0 \leq i < n$ and the second element is of type τ_i. We can use $choose(x, \tau_0, \dots, \tau_{n-1})$ to stand for a type which must be one of $\tau_0, \dots, \tau_{n-1}$, determined by the value of x: the type is τ_i if $x = i$. Also we present some additional rules in Figure 11 for handling type coercion involving sum types (rules for type equality are omitted).

Now we can define $sum(\tau_0, \dots, \tau_{n-1})$ as:

$$\exists a : nat_n.\mathrm{int}(a) * choose(a, \tau_0, \dots, \tau_{n-1}),$$

that is, a value of type $sum(\tau_0, \dots, \tau_{n-1})$ is represented as a pair in which the first part is a tag determining the type of the second part. We present an example to illustrate the use of sum types.

In Figure 12, we declare a dependent datatype in Xanadu for lists; `Nil` is given the type `<'a> list(0)`, that is, it is a list of length 0; `Cons` is assigned the type

```
{n:nat} 'a * <'a> list(n) -> <'a> list(n+1),
```

indicating that `Cons` takes an element and a list of length n and yields a list of length $n + 1$. This leads us to represent the type constructor $list$ as follows,

$$\mu t.\Lambda\alpha.\Pi n : nat.(\exists\phi_0.unit) + (\exists\phi_1.\alpha * (\alpha)t(a)),$$

where μ is the fixed point operator and ϕ_0 is $n = 0$ and ϕ_1 is $a : nat, a + 1 = n$. If we unfold $(\tau)list(n)$, we obtain the type $(\exists\phi_0.unit) + (\exists\phi_1.\tau * (\tau)list(a))$, which can be folded into $(\tau)list(n)$. It is straightforward to apply this strategy to a general case of dependent datatypes. We provide two auxiliary instructions `fold`$[\tau]$ r and `unfold` r to indicate the need for folding the type of r into τ and unfolding the type of r, respectively.

The DTAL code in Figure 13 corresponds to the Xanadu program in Figure 12. The state type following the label `length` indicates that the top element on the stack is a list and the second one is a label; the list is the argument of the function and the label is the return address (pushed onto the stack by the caller); the type of the label states that the top element of the stack is an integer, which is to be the return value of the function, and the rest of the stack is the same as the current stack excluding the top two elements. The state type following the label `length` precisely indicates that this is a function that accepts a list of length n and return an integer of value n. We regard the representation of dependent datatypes at assembly level as a significant contribution, which makes it possible to perform compilation with dependent types for programs in DML and thus certify more program properties.

The DTAL code in Figure 13 is unsatisfactory for the following reason. In practice, the list constructors are usually represented without tags for both efficiency and memory concern. In other words, we can interpret $(\alpha)list$ as $\exists a : nat_2.choose(a, unit, \alpha * (\alpha)list)$. The reason is that it can be readily tested in practice whether a value equals $\langle\rangle$ (which is commonly represented as a null pointer), and therefore there is no need for a tag. This optimized list representation can also be handled in DTAL. Please see [16] for details.

The treatment of sum types extends the one in [3]. There indexed sums $\tau_1 +_i \tau_2$ $(i = 1, 2)$ are introduced for types τ_1 and τ_2 in addition to the standard sum $\tau_1 + \tau_2$. The typing rules for indexed sums essentially state that for $i = 1, 2$, $in_i(e) : \tau_1 +_i \tau_2$ is derivable if $e : \tau_i$ is, where in_i is used to indicate which rule is applied. To relate indexed sums to sum, there are subtyping rules for making $\tau_1 +_i \tau_2$ a subtype of $\tau_1 + \tau_2$ for $i = 1, 2$. In DTAL, $\tau_1 +_i \tau_2$ can be interpreted as $\mathrm{int}(i-1) * choose(i-1, \tau_1, \tau_2)$ and the subtyping relation can be derived with the use of type coercion rules.

7. IMPLEMENTATION

We have prototyped a type-checker and an interpreter for DTAL and verified many examples, providing a proof of concept. The implementation and examples are available on-line [14].

We have also prototyped a compiler which produces DTAL code from source programs in *Xanadu*, a language with C-like syntax in which only top level functions are supported and no pointers are allowed. Xanadu shares many common features with languages like Safe C [9] and Popcorn [6]. The most significant feature of Xanadu is its type system, which supports a restricted form of dependent types that are similar to those in DTAL, though registers are replaced with local variables in a program. Please see [15] for more details.

```
length: ('r, 'a){n:nat} [sp: 'a list(n) :: [sp: int(n) :: 'r] :: 'r]
        // [sp: int(n) :: 'r] represents the state type of the return
        //  address (label) which is pushed on the stack by the caller.
        // Note that 'a list is represented as a dependent type internally
        pop     r1          // pop the list argument into r1
        mov     r2, 0       // initialize r2

loop:   ('r, 'a){i:nat, j:nat | i+j=n} [r1: 'a list(i), r2: int(j), sp: [sp: int(n) :: 'r] :: 'r]
        unfold  r1          //
        load    r3, r1(0)   // load list tag into r3 (r3 = 0 or 1)
        beq     r3, finish  // goto finish if r1 is empty (r3 = 0)
        load    r1, r1(1)   // r1: 'a * 'a list(i-1) (r3 = 1 since r3 is not 0)
        load    r1, r1(1)   // move list tail into r1
        add     r2, r2, 1   // r2: int(j+1)
        jmp     loop        // loop again

finish: ('r){n:nat} [r2: int(n), sp: [sp: int(n) :: 'r] :: 'r]
        pop     r1  // return address pops into r1
        push    r2  // result pushes onto the stack
        jmp     r1  // return
```

Figure 13: An implementation of the length function on lists in DTAL

The compilation is like compiling C into a typical untyped assembly language except that here we need to construct state types for labels. We have compiled all the examples in this paper.[2]

In Xanadu, we allow the programmer to provide loop invariants in the form of dependent types so that significantly more array bound checks can be eliminated in practice. In Figure 14, the top part is a program in Xanadu, which initializes an array with zeros, and the rest is the DTAL code compiled from the program. The function header:

```
{n:nat} unit initialize(int vec[n])
```

indicates that for every natural number n, `initialize` takes an integer array of size n and returns no value. The type following the keyword `invariant` essentially states that `i` and `l` are of types $\mathrm{int}(a)$ and $\mathrm{int}(b)$, respectively, where a and b are natural numbers satisfying $a+b=n$. Note that n is the size of array `vec`.

The Xanadu program can be compiled into the DTAL code excluding the state types for labels in a standard manner. This part is exactly like compiling a corresponding C program. We briefly mention the construction of the state types in Figure 14. Notice that the state type attached to `loop` is essentially translated from the type annotation in the source program. We simply modify the annotation to include the types of variables not mentioned and then replace the variables with the registers to which these variables are mapped. We expect to formalize such a compilation strategy in future and show that a well-typed Xanadu program can always be thus compiled into well-typed DTAL code. At present, we may merely view the type annotations in Xanadu as compilation hints to generating well-typed DTAL code.

8. RELATED WORK

There is a great deal of ongoing research on certifying compilers. Examples of certifying compilers for type and memory safety include various ones compiling Java into Java virtual machine language (JVML), Touchstone compiling Safe C into a form of proof-carrying code (which we call TPCC) [9], TIL [11] and its successor TILT and FLINT/ML [10] compiling SML [5] into a typed intermediate language [11], and ROML [12] compiling a restricted set of ML into a portion of C that is type safe.

DTAL is an extension of TAL with dependent types, and it can be readily transformed into a TAL-like language if one erases all syntax related to type index expressions. In this respect, DTAL generalizes TAL. In DTAL, initialization is treated differently from in TAL. A type in TAL can be annotated with a flag to indicate the initialization status of a value with this type, but the type *top* is used in DTAL to represent the type of all uninitialized values. This strategy works because every array (and tuple if presented) is initialized upon allocation in DTAL.

The notion of proof-carrying code introduced in [8] can address the memory safety issue in mobile code as follows. The essential idea is to generate a proof asserting the memory safety property of code and then attach it to the code. The proof carried by the code can then be verified before execution. This is an attractive approach but a challenging question remains, that is, how to generate a proof to assert memory safety property of a (large and complex) program. The Touchstone compiler [9], which compiles programs written in a type-safe subset of C into proof-carrying code (TPCC for Touchstone's PCC), handles this question through a general verification condition generator [1], generating verification conditions for both type safety and memory safety. Also TPCC performs some loop invariant synthesis for eliminating array bound checks. In general, TPCC seems more involved in handling type safety when compared to TAL, while TAL seems less flexible than TPCC.

DML is a functional programming language that enriches ML with a restricted form of dependent types [18], allow-

[2]We currently do not have a pretty printer for the generated DTAL code, and therefore we took the liberty to prettify the DTAL code presented in this paper.

```
{n:nat} unit initialize(int vec[n]) {
   var: int i, l;;
   i = 0; l = arraysize(vec);
   invariant: [a:nat, b:nat | a + b = n] (i: int(a), l: int(b))
   while (l > 0) { vec[i] = 0; i = i + 1; l = l - 1; }
}

init:   ('r) {n:nat} [sp: int array(n) :: [sp: 'r] :: 'r]
        pop        r1
        mov        r2, 0
        arraysize  r3, r1

loop:   ('r) {n:nat, a:nat, b:nat | a + b = n}
        [r1: int array(n), r2: int(a), r3: int(b), sp: [sp: 'r] :: 'r]
        blte       r3, finish
        store      r1(r2), 0
        add        r2, r2, 1
        sub        r3, r3, 1
        jmp        loop

finish: ('r) [sp: [sp: 'r] :: 'r]
        pop  r1
        jmp  r1
```

Figure 14: Implementations of an initialization function in Xanadu and DTAL

ing the programmer to capture more program invariants through types and thus to detect more program errors at compile-time. In particular, the programmer can capture more invariants in data structures by refining datatypes with type index expressions. For instance, one can form a datatype in DML that is precisely for all red/black trees and program with such a type. The type system of DML is also studied for array bound check elimination [17].

DTAL stands as an alternative design choice to TPCC, extending TAL with a form of dependent types that is largely adopted from DML. The design of DTAL is partly motivated by an attempt to build a certifying compiler for DML. Unlike TPCC, there are no proofs attached to DTAL code. The verifier for DTAL code is a dependent type-checker consisting of a constraint generator and a constraint solver. In general, proof verification is easier than proof search, and therefore the TPCC startup overhead should be less than that for DTAL code, though it seems too difficult at this stage to perform a meaningful comparison. In future, we are also interesting in constructing a proof asserting the well-typedness of DTAL code and thus provide a means to generating a form of proof-carrying code from programs in Xanadu. This is appealing as Xanadu allows the programmer to formally supply program invariants that may be too sophisticated to synthesize and thus facilitates the construction of proof-carrying code.

We view DTAL as a type-theoretic approach to reasoning about memory safety at assembly level. With a stronger type system than that of TAL, DTAL is expected to capture program errors that can slip through the type system of TAL. This is supported by the fact that DML can capture program errors in practice which eludes the type system of ML.

9. CONCLUSION

TAL is a typed assembly language with a type system at assembly level. The type system of TAL contains some limitations that prevent certain important loop-based optimizations such as array bound check elimination and tag check elimination. We have enriched TAL with a restricted form of dependent types and the enrichment leads to a dependently typed assembly language (DTAL) that overcomes these limitations. We have established the soundness of the type system of DTAL and implemented a type-checking algorithm. We have also constructed a prototype compiler which compiles Xanadu programs into DTAL, where Xanadu is a programming language with C-like syntax that supports a dependent type system similar to that of DTAL but significantly more involved.

In future work, we intend to study compilation with dependent types, translating programs in DML into DTAL. We feel that the presented approach to representing dependent datatypes in DTAL has made a significant step towards achieving this goal. On a larger scale, we are interested in both using types to capture more program properties in high-level languages and constructing certifying compilers to translate these properties into low-level languages.

10. ACKNOWLEDGMENT

We thank the anonymous referees for their detailed constructive comments, which have undoubtedly raised the quality of the paper.

11. REFERENCES

[1] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[2] R. Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51:201–206, 1994.

[3] R. Harper and C. Stone. A type-theoretic interpretation of Standard ML. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Robin Milner Festschrifft*. MIT Press, 1998. (To appear).

[4] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, Italy, 1984.

[5] R. Milner, M. Tofte, R. W. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1997.

[6] G. Morrisett et al. Talx86: A realistic typed assembly language. In *Proceedings of Workshop on Compiler Support for System Software*, 1999.

[7] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 85–97, January 1998.

[8] G. Necula. Proof-carrying code. In *Conference Record of 24th Annual ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM press, 1997.

[9] G. Necula and P. Lee. The design and implementation of a certifying compiler. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 333–344. ACM press, June 1998.

[10] Z. Shao. An Overview of the FLINT/ML compiler. In *Proceedings of ACM SIGPLAN Workshop on Types in Compilation (TIC '97)*, June 1997.

[11] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. A type-directed optimizing compiler for ML. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, June 1996.

[12] A. Tolmach and D. P. Oliva. From ML to Ada(!?!): Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.

[13] H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998. pp. viii+189. Available as `http://www.cs.cmu.edu/~hwxi/DML/thesis.ps`.

[14] H. Xi. Implementations and Examples for Xanadu and DTAL. Available at `http://www.ececs.uc.edu/~hwxi/Xanadu-DTAL`, 1999.

[15] H. Xi. Imperative Programming with Dependent Types. In *Proceedings of 15th IEEE Symposium on Logic in Computer Science*, pages 375–387, June 2000.

[16] H. Xi and R. Harper. A Dependently Typed Assembly Language. Technical Report CSE-99-008, Oregon Graduate Institute, July 1999. Also available as `http://www.ececs.uc.edu/~hwxi/academic/papers/DTAL.ps`.

[17] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.

[18] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.