A Dependently Typed Assembly Language

Hongwei Xi
Department of Computer Science and Engineering
Oregon Graduate Institute of Science and Technology
hongwei@cse.ogi.edu

Robert Harper
Department of Computer Science
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 assembly level. DTAL overcomes several significant limitations in recently proposed low-level languages including Java bytecode language and a typed assembly language, which prevent them from handling certain important compiler optimizations such as run-time array bound check elimination. We also mention a compiler which can generate DTAL code from compiling some high-level programs.

1 Introduction

A certifying compiler is one that generates object code that can be readily checked for compliance with a specified safety policy that constrains its run-time behavior. By ensuring that compliance is checkable, the code recipient need not be concerned with the origin of the code, only the (augmented) code itself. Typical safety policies include type safety (which excludes, for examples, programs that attempt to add an integer to a floating point number) and memory safety (which excludes stray memory accesses). Examples of certifying compilers for this property 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) (Necula and Lee 1998), TIL and its successor TILT compiling Standard ML (Milner, Tofte, Harper, and MacQueen 1997) into a typed intermediate language (Tarditi, Morrisett, Cheng, Stone, Harper, and Lee 1996), and ROML compiling a restricted set of ML into a portion of C that is type safe (Tolmach and Oliva 1998).

Certifying compilers have a number of benefits, including facilitating safe exchange of code in an untrusted environment and improving the robustness of a compiler (by thinking of each transformation phase as a separate certifying compiler whose subsequent stages may check compliance with some safety policy). Specific approaches to certification include proof-carrying code (Necula 1997) (adopted in Touchstone), in which both type safety and memory safety are expressed by (first-order) logic assertions about program variables, is checked by a VC generator and a theorem prover and code is certified by an explicit representation of the proof; and type systems (adopted in TIL), in which 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, but it is yet to be studied whether this approach can readily extend to higher-order languages. 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.

A typed assembly language (TAL) is formed in (Morrisett, Walker, Crary, and Glew 1998), 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.

The goal of this paper is to 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, 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 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 (Xi and Pfenning 1999), we introduce a clear separation between ordinary run-time expressions and a distinguished family of index expressions, linked by singleton types of form $int(x)$: every integer expression of type $int(x)$ 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 $int(x)$ (written as $e : int(x)$) involves non-trivial equational reasoning about the run-time behavior of e. For instance, $e : int(3)$ means that e, when evaluated, must evaluates to 3. Clearly, $3 : int(3)$, and perhaps, $1 + 2 : int(3)$, but it is, in general, undecidable whether an arbitrary (possibly effectful) e has type $int(x)$. This is where theorem proving / constraint satisfaction comes into the picture. A crucial feature in DML, which is to be adopted in this paper, is the use of existential dependent types, which makes it possible to avoid difficult constraints and type realistic programs.

We have formed a dependently typed assembly language (DTAL) that supports a limited form of dependent type system capturing both type safety and memory safety. We have also designed a language *Xanadu* with C-like syntax and prototyped a compiler that compiles Xanadu into DTAL. This paper concentrates on DTAL, though we occasionally use programs in Xanadu notation to facilitate the presentation of some notions in DTAL.

The Xanadu program in Figure 1 implements a copy function on arrays. Notice that the simplicity of this example is solely for the sake of illustration purpose and should not be interpreted as the limitation of our approach. 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 `{m:nat, n:nat | m <= n}` is a universal quantifier as is explained in (Xi and Pfenning 1998) and `int src[m]` (`int dst[n]`) means that `src` (`dst`) is an integer array of size `m` (`n`). We use `var:` to start variable declaration, which ends with `;;`. Also the function `arraysize` returns the size of an array.

The DTAL code in Figure 2 basically corresponds to the Xanadu program. A double slash `//` starts a comment line. 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

```
{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

```
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 as shown later
```

Figure 2: A copy function implemented in DTAL

element 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. 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.

The main contribution of the paper is a formulation of a dependent type system for an (imperative) assembly-level language 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, and (d) provides an approach to certification based on type-checking. One trade-off 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.

We will also briefly mention certain aspects on a compilation from Xanadu into DTAL, which allows us to construct a toy compiler for generating sample DTAL code. The details on Xanadu can be found in (Xi 1999a) while the compilation from Xanadu or other higher-level languages into DTAL is to be reported in future work.

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 and proven 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 (Necula and Lee 1998) and Popcorn (Morrisett et al. 1999) with C-like syntax, into DTAL. The rest of the paper discusses some closely related work and future directions.

2 Dependently Typed Assembly Language

In this section we present a typed assembly language in which a restricted form of dependent types is available. This closely relates to the typed assembly languages given in (Morrisett, Walker, Crary, and Glew 1998; Morrisett, Crary, Glew, and Walker 1998), but there are also many substantial differences which we will point out. We use the name DTAL for this dependently typed assembly language.

2.1 Syntax

In this paper, we do not address the stack overflow issue. We assume that there are a fixed number n_r of registers and a stack of infinite depth. A *type state* Σ consists of a pair (R, S), where R is a finite mapping from the set $\{0, 1, \ldots, n_r - 1\}$ into types and S is a stack type. The intention is to capture some type information on the register file and stack with R and S, respectively. The syntax for DTAL is given in Figure 3.

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 in this paper. The justification for this choice is that we have previously used this domain to eliminate array bound checks effectively (Xi and Pfenning 1998).

type variables	α		
type states	Σ	::=	(R, S)
state types	σ	::=	$\mathit{state}(\lambda\Delta.\lambda\phi.\Sigma)$
regfile types	R	::=	$[r_0 : \tau_0, \ldots, r_{n_r-1} : \tau_{n_r-1}]$
stack types variables	ρ		
stack types	S	::=	$[] \mid \rho \mid \tau :: S$
types	τ	::=	$\alpha \mid \sigma \mid \mathit{top} \mid \mathit{int}(x) \mid \tau\ \mathit{array}(x) \mid$ $\mathit{prod}(\tau_1, \ldots, \tau_n) \mid \exists a : \gamma.\tau$
type erasures	ϵ	::=	$\alpha \mid \mathit{top} \mid \mathit{int} \mid \epsilon\ \mathit{array} \mid \mathit{prod}(\epsilon_1, \ldots, \epsilon_n)$
type variable contexts	Δ	::=	$\cdot_{\mathrm{tv}} \mid \Delta, \rho \mid \Delta, \alpha$
registers	r	::=	$r_0, \ldots, r_{n_r-1}$
instructions	ins	::=	$\mathit{aop}\ r_d, r_s, v \mid \mathit{bop}\ r, v \mid \texttt{arraysize}\ r_d, r_s \mid$ $\texttt{mov}\ r, v \mid \texttt{load}\ r_d, r_s(v) \mid \texttt{store}\ r_d(v), v_s \mid$ $\texttt{newtuple}[\tau]\ r \mid \texttt{newarray}[\tau]\ r$ $\texttt{jmp}\ v \mid \texttt{pop}\ r \mid \texttt{push}\ v \mid \texttt{halt}$
constants	c	::=	$\langle\rangle \mid i \mid l$
values	v	::=	$c \mid r$
instruction sequences	I	::=	$\texttt{jmp}\ v \mid \texttt{halt} \mid \mathit{ins}; I$
blocks	B	::=	$\lambda\Delta.\lambda\phi.(\Sigma, I)$
arithmetic ops	aop	::=	$\texttt{add} \mid \texttt{sub} \mid \texttt{mul} \mid \texttt{div}$
branch ops	bop	::=	$\texttt{beq} \mid \texttt{bne} \mid \texttt{blt} \mid \texttt{blte} \mid \texttt{bgt} \mid \texttt{bgte}$
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

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. We also use certain transparent abbreviations, such as $0 \leq x < y$ which stands for $0 \leq x \wedge x < y$. 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 $\{a : \mathit{int} \mid a \geq 0\}$.

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 $\mathit{int}(i_1)$

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

Figure 4: Syntax for type index expressions

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

Figure 5: Some explanation on the program in Figure 2

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 $prod(\tau_1, \ldots, \tau_n)$ for the product of types $\tau_1, \ldots, \tau_n$, which is usually written as $\tau_1 * \cdots * \tau_n$. This notation allows us to clearly distinguish τ from $prod(\tau)$. Also we use *unit* for the empty product $prod()$. Nonetheless, we may use the notation $\tau_1 * \cdots * \tau_n$ if it risks no confusion. We use *top* for the type of uninitialized registers and assume that a register is initialized if it is not of type *top*.[1]

A block $B = \lambda\Delta\lambda\phi.(\Sigma, I)$ roughly means that B is polymorphic on Δ and ϕ. In order to execute the block on an abstract machine, we need to find substitutions Θ and θ for Δ and θ, respectively, such that the current machine state entails the state $\Sigma[\Theta][\theta]$ and then execute $I[\Theta][\theta]$. The entailment of a state Σ basically means that the type assignment to registers and stack in Σ correctly reflects the types of registers and stack in the current abstract machine. For instance, if Σ indicates that an integer is on top of the stack, then an integer must be stored on top of the stack in the abstract machine.

A state type $\Sigma(\lambda\Delta.\lambda\phi.\Sigma)$, 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 $\Sigma[\Theta][\theta]$ whenever the execution reaches the label. Note that *state* is a type constructor for constructing state types.

We use J for a general instruction sequence in the following presentation, which consists of a sequence of instructions or labels. Given a block $B = \lambda\Delta.\lambda\phi.(\Sigma, I)$, we write $\sigma(B)$ for $state(\lambda\Delta.\lambda\phi.\Sigma)$ 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

[1] We could not use the type *unit* for this purpose since we always represent a value of type *unit* as a null pointer so that we can implement data structures such as linked list.

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. The convention is that we may omit quantifying over stack type variables in the concrete syntax; 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 its domain. Note that we write int for $\exists a : \text{int}.int(a)$, 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.

$$\|top\| = top \qquad \|\alpha\| = \alpha \qquad \|int(x)\| = int$$
$$\|\sigma\| = unit \qquad \|\tau\ array(x)\| = \|\tau\|\ array$$
$$\|prod(\tau_1, \ldots, \tau_n)\| = prod(\|\tau_1\|, \ldots, \|\tau_n\|)$$
$$\|\exists a : \gamma.\tau\| = \|\tau\|$$

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.

2.2 Dynamic Semantics

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

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 constant or a heap address.

Given $i \in \mathbf{dom}(\mathcal{R})$, $\mathcal{R}(i)$ is either a heap address or a constant. Also $\mathcal{S}$ is always a list of form $hc_0 :: \ldots :: hc_{n-1} :: []$ for some heap addresses or constants $hc_0, \ldots, hc_{n-1}$, where we use $::$ for the list constructor and $[]$ for the empty list. We write `sp` for the stack pointer which always points to the top of the stack. Given $\mathcal{S} = hc_0 :: \ldots :: hc_{n-1} :: []$, we write $\mathtt{sp}(i)$ for hc_i, where $i = 0, \ldots, n-1$.

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 $(ic, \mathcal{M}) \rightarrow_P (ic', \mathcal{M}')$ 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'. The evaluation rules for DTAL are presented in Figure 6. We do not consider garbage collection in this abstract machine, and therefore the heap can only be affected by two memory allocation instructions `newtuple` and `newarray`.

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

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

$$\frac{\begin{array}{c}\|\tau\| = \mathit{prod}(\epsilon_1,\ldots,\epsilon_n) \quad J(ic) = \mathtt{newtuple}[\tau]\ r \\ h \notin \mathbf{dom}(\mathcal{H}) \quad \mathcal{H}' = \mathcal{H}[h \mapsto (hc_1,\ldots,hc_n)]\end{array}}{(ic,(\mathcal{H},\mathcal{R},hc_1 :: \ldots :: hc_n :: \mathcal{S})) \to_P (ic+1,(\mathcal{H}',\mathcal{R}[r=h],\mathcal{S}))}\ \textbf{(eval-newtuple)}$$

$$\frac{\|\tau\| = (\epsilon)\mathit{array} \quad J(ic) = \mathtt{newarray}[\tau]\ r \quad n \geq 0 \quad h \notin \mathbf{dom}(\mathcal{H})}{(ic,(\mathcal{H},\mathcal{R},n :: hc :: \mathcal{S})) \to_P (ic+1,(\mathcal{H}[h \mapsto (hc,\ldots,hc)],R[r=h],\mathcal{S}))}\ \textbf{(eval-newarray)}$$

$$\frac{J(ic) = \mathtt{arraysize}\ r_d,r_s \quad \mathcal{H}(\mathcal{M}(r_s)) = (hc_0,\ldots,hc_{n-1})}{(ic,(\mathcal{H},\mathcal{R},\mathcal{S})) \to_P (ic+1,(\mathcal{H},R[r=n],\mathcal{S}))}\ \textbf{(eval-arraysize)}$$

$$\frac{J(ic) = \mathtt{add}\ r_d,r_s,v \quad \mathcal{M}(r_s) = i \quad \mathcal{M}(v) = j}{(ic,\mathcal{M}) \to_P (ic+1,\mathcal{M}[r_d = i+j])}\ \textbf{(eval-add)}$$

$$\frac{J(ic) = \mathtt{sub}\ r_d,r_s,v \quad \mathcal{M}(r_s) = i \quad \mathcal{M}(v) = j}{(ic,\mathcal{M}) \to_P (ic+1,\mathcal{M}[r_d = i-j])}\ \textbf{(eval-sub)}$$

$$\frac{J(ic) = \mathtt{mul}\ r_d,r_s,v \quad \mathcal{M}(r_s) = i \quad \mathcal{M}(v) = j}{(ic,\mathcal{M}) \to_P (ic+1,\mathcal{M}[r_d = i*j])}\ \textbf{(eval-mul)}$$

$$\frac{J(ic) = \mathtt{div}\ r_d,r_s,v \quad \mathcal{M}(r_s) = i \quad \mathcal{M}(v) = j \quad j \neq 0}{(ic,\mathcal{M}) \to_P (ic+1,\mathcal{M}[r_d = i \div j])}\ \textbf{(eval-div)}$$

$$\frac{J(ic) = \mathtt{beq}\ r,v \quad \mathcal{M}(r) = 0 \quad \mathcal{M}(v) = l}{(ic,\mathcal{M}) \to_P (J^{-1}(l)+1,\mathcal{M})}\ \textbf{(eval-beq-true}$$

$$\frac{J(ic) = \mathtt{beq}\ r,v \quad \mathcal{M}(r) \neq 0}{(ic,\mathcal{M}) \to_P (ic+1,\mathcal{M})}\ \textbf{(eval-beq-false)}$$

$$\frac{J(ic) = \mathtt{bne}\ r,v \quad \mathcal{M}(r) \neq 0 \quad \mathcal{M}(v) = l}{(ic,\mathcal{M}) \to_P (J^{-1}(l)+1,\mathcal{M})}\ \textbf{(eval-bne-true)}$$

$$\frac{J(ic) = \mathtt{bne}\ r,v \quad \mathcal{M}(r) = 0}{(ic,\mathcal{M}) \to_P (ic+1,\mathcal{M})}\ \textbf{(eval-bne-false)}$$

$$\frac{J(ic) = \mathtt{jmp}\ v \quad \mathcal{M}(v) = l}{(ic,\mathcal{M}) \to_P (J^{-1}(l)+1,\mathcal{M})}\ \textbf{(eval-jmp)}$$

$$\frac{J(ic) = \mathtt{mov}\ r,v \quad \mathcal{M}(v) = hc}{(ic,\mathcal{M}) \to (ic+1,\mathcal{M}[r = hc])}\ \textbf{(eval-mov)}$$

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

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

$$\frac{J(ic) = \mathtt{pop}\ r}{(ic,(\mathcal{H},\mathcal{R},hc :: \mathcal{S})) \to (ic+1,(\mathcal{H},\mathcal{R}[r = hc],\mathcal{S}))}\ \textbf{(eval-pop)}$$

$$\frac{J(ic) = \mathtt{push}\ v \quad \mathcal{M}(v) = hc}{(ic,(\mathcal{H},\mathcal{R},\mathcal{S})) \to (ic+1,(\mathcal{H},\mathcal{R},hc :: \mathcal{S}))}\ \textbf{(eval-push)}$$

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

Figure 6: Evaluation rules

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

We present the evaluation rules for most instructions and the rest can be readily constructed from the closely related ones. 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 $0 \leq i < n$ always holds when either **(eval-load)** or **(eval-store)** is 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). In order to perform array bound checking efficiently, we can always group each array with its size after allocation.

It should be stressed that there is no interaction between the dynamic semantics of DTAL and type index expressions. Therefore, erasing type index expressions in a program can not alter the execution behavior of the program. We have omitted the evaluation rules for `blt`, `blte`, `bgt`, `bgte`, which can be readily formulated by following the rules for `beq` and `bne`.

Notice that the instruction `jmp` v jumps to the instruction immediately following l, if v evaluates to l under the current machine state. Also notice that the rules **(eval-newtuple)** and **(eval-newarray)** are non-standard. If $\|\tau\|$ is of form $prod(\epsilon_1, \ldots, \epsilon_n)$, then `newtuple`$[\tau]$ r allocates n *new* word memory on heap and stores a pointer in r which points to the allocated memory. Then it moves the content in $\mathtt{sp}(i)$ into $r(i)$ for $i = 0, \ldots, n-1$ and decrease `sp` by n. For example, the follow code allocates a pair on heap and stores the pointer into `r1`, which ends with type $\exists a : nat.prod(int(a), int(a))$, that is, a pair of integers of the same value. Note that we use `{a:nat}` as the concrete syntax for $\exists a : nat$ in DTAL code.

```
mov                                 r1, 0
push                                r1
push                                r1
newtuple[{a:int} (int(a) * int(a))] r1
```

Similarly, if $\|\tau\|$ is of form $(\epsilon)array$, then `newarray`$[\tau]$ r allocates n new word memory on heap, where n is the value of the integer stored in $sp[0]$, and stores a pointer in r which points to the allocated memory. Then it copies the content in $\mathtt{sp}[1]$ into $r(0), \ldots, r(n-1)$, and decrease `sp` by 2. For example, the following code allocates an array of type $(\exists a : nat.int(a))array(1024)$, that is, an array of size 1024 whose elements are natural numbers.

```
mov                         r1, 1024
push                        r1
mov                         r1, 0
push                        r1
newarray[{a:nat}int(a)] r1
```

We emphasize that h must be new in both rules **(eval-newtuple)** and **(eval-newarray)**, that is, h is not already in the domain of $\mathcal{H}$. The typing consequences of these malloc instructions are explained in the next section, where the typing rules **(type-newtuple)** and **(type-newarray)** are introduced.

Definition 2.1 *Given a program P, Q is defined as a P-snapshot if $(0, \mathcal{M}_0) \rightarrow_P^* Q$ holds for some machine state $\mathcal{M}_0$, where $\rightarrow_P^*$ is the reflexive and transitive closure of $\rightarrow_P$. We say that a program P is* well-structured *if every P-snapshot which is not* `HALT` *evaluates to another P-snapshot.*

In other words, the evaluation of a well-structured program is never stuck. Notice that a well-structured program is both type-safe and memory-safe according to the evaluation rules for DTAL. Certainly it is undecidable to precisely determine 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 type system which supports a restricted form of dependent types. This approach can accept highly optimized programs such as the binary search example in Figure 17, where all array bound checks are removed.

2.3 Static Semantics

We present the typing rules for DTAL in this section. Given a type state $\Sigma = (R, S)$, we use an array representation for R and a list representation for S, where the list for S always ends with a stack variable ρ.

In the presence of dependent types, it is no longer trivial whether a type τ is well-formed. For instance, we must disallow the occurrence of a type like *int array*(-1) in the typing rules for DTAL since it can readily lead to inconsistency of the type system. In other words, we must prove that x is a natural number when forming the type *int array*(x).

We present type formation rules in Figure 7, where we write $\phi; \Delta \vdash \tau : *$ to mean that τ is a well-formed type under context $\phi; \Delta$. Similarly, we write $\phi; \Delta \vdash \Sigma[\text{well-formed}]$ to mean that Σ is well-formed under context $\phi; \Delta$. The well-formedness of types and type states can be derived through the application of these rules. From now on, we assume that all types and type states are well-formed in the following presentation.

We use a judgment of form $\phi; \Delta; \Sigma \vdash_\Lambda v : \tau$ to mean that value v is assigned type τ under the context $\phi; \Delta; \Sigma$ 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 8 are for typing unit, integers, labels and registers.

We present the typing rules for DTAL in Figure 9 and Figure 10. A judgment of form $\phi; \Delta; \Sigma \vdash I$ means that the instruction sequence I is well-typed under context $\phi; \Delta; \Sigma$. The

$$\frac{\phi \vdash x : int}{\phi; \Delta \vdash int(x) : *} \qquad \frac{\phi; \Delta \vdash \tau_1 : * \quad \cdots \quad \phi; \Delta \vdash \tau_n : *}{\phi; \Delta \vdash prod(\tau_1, \ldots, \tau_n) : *} \qquad \frac{\phi \vdash x : nat; \quad \phi; \Delta \vdash \tau : *}{\phi; \Delta \vdash \tau\ array(x) : *}$$

$$\frac{\phi, \phi'; \Delta, \Delta' \vdash \Sigma[\text{well-formed}]}{\phi; \Delta \vdash state(\lambda\Delta'.\lambda\phi'.\Sigma) : *} \qquad \frac{\phi, a : \gamma; \Delta \vdash \tau : *}{\phi; \Delta \vdash \exists a : \gamma.\tau : *}$$

$$\frac{}{\phi; \Delta \vdash \rho[\text{well-formed}]} \qquad \frac{\phi \vdash x : int \quad \phi; \Delta \vdash \tau : * \quad \phi; \Delta \vdash S[\text{well-formed}]}{\phi; \Delta \vdash \tau :: S[\text{well-formed}]}$$

$$\frac{\phi; \Delta \vdash R(i) : * \text{ for } 0 \leq i < n_r}{\phi; \Delta \vdash R[\text{well-formed}]} \qquad \frac{\phi; \Delta \vdash R[\text{well-formed}] \quad \phi; \Delta \vdash S[\text{well-formed}]}{\phi; \Delta \vdash (R, S)[\text{well-formed}]}$$

Figure 7: Type formation rules for DTAL

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

$$\frac{\Lambda(l) = \sigma}{\phi; \Delta; (R, S) \vdash_\Lambda l : \sigma}\ \textbf{(type-label)} \qquad \frac{0 \leq i < n_r}{\phi; \Delta; (R, S) \vdash_\Lambda r_i : R(i)}\ \textbf{(type-reg)}$$

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

Figure 8: Typing rules for integers, labels, registers and stack cells.

notation $R[r : \tau]$ means that we update the type of register r to τ in Σ, that is, if r is the ith register, then we update the content of $R(i)$ with τ.

The rules **(type-newtuple)** and **(type-newarray)** are for typing tuples and arrays allocated on heap, respectively. We have explained in the previous section how memory allocation is performed.

We give some explanation on the rule **(type-beq)**. Suppose that we type-check `beq`; $r, v; I$ under $\phi; \Delta; \Sigma$; we first check that r has type $int(x)$ for some x; we then type-check I under $\phi, x \neq 0; \Delta; \Sigma$ ($x \neq 0$ is added into ϕ since the jump is not taken in this case); we also verify that v has a state type and $\phi, x = 0; \Delta; \Sigma$ 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 $\phi; \Delta; \Sigma$, and we have verified that r_s and v have types $\tau\ array(x)$ and $int(y)$, respectively, and we can prove $\phi \models 0 < x$ 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, which is similar and thus omitted.

We use $\vdash P[\text{well-typed}]$ to mean that a program $P = (l_1 : B_1, \ldots, l_n : B_n)$ is well-typed,

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

$$\frac{\phi; \Delta; (R[r:\tau], S) \vdash I \quad \phi; \Delta \models \mathit{prod}(\tau_0, \ldots, \tau_{n-1}) \leq \tau}{\phi; \Delta; (R, \tau_0 :: \ldots :: \tau_{n-1} :: S) \vdash \mathtt{newtuple}[\tau]\ r; I} \ \textbf{(type-newtuple)}$$

$$\frac{\phi \models x \geq 0 \quad \phi; \Delta \vdash \tau_1 \leq \tau \quad \phi; \Delta; (R[r:\tau\ \mathit{array}(x)], S) \vdash I}{\phi; \Delta; (R, \mathit{int}(x) :: \tau_1 :: S) \vdash \mathtt{newarray}[\tau]\ r; I} \ \textbf{(type-newarray)}$$

$$\frac{\phi; \Delta \vdash r_s : \tau\ \mathit{array}(x) \quad \phi; \Delta; (R[r_d : \mathit{int}(x)], S) \vdash I}{\phi; \Delta; (R, S) \vdash \mathtt{arraysize}\ r_d, r_s; I} \ \textbf{(type-arraysize)}$$

$$\frac{\phi; \Delta; (R, S) \vdash v : \tau \quad \phi; \Delta; (R[r:\tau], S) \vdash I}{\phi; \Delta; (R, S) \vdash \mathtt{mov}\ r, v; I} \ \textbf{(type-mov)}$$

$$\frac{\begin{array}{c}\phi; \Delta; (R, S) \vdash r_s : \mathit{int}(x) \quad \phi; \Delta; (R, S) \vdash v : \mathit{int}(y) \\ \phi; \Delta; (R[r_d : \mathit{int}(x+y)], S) \vdash I\end{array}}{\phi; \Delta; (R, S) \vdash \mathtt{add}\ r_d, r_s, v; I} \ \textbf{(type-add)}$$

$$\frac{\begin{array}{c}\phi; \Delta; (R, S) \vdash r_s : \mathit{int}(x) \quad \phi; \Delta; (R, S) \vdash v : \mathit{int}(y) \\ \phi; \Delta; (R[r_d : \mathit{int}(x-y)], S) \vdash I\end{array}}{\phi; \Delta; (R, S) \vdash \mathtt{sub}\ r_d, r_s, v; I} \ \textbf{(type-sub)}$$

$$\frac{\begin{array}{c}\phi; \Delta; (R, S) \vdash r_s : \mathit{int}(x) \quad \phi; \Delta; (R, S) \vdash v : \mathit{int}(y) \\ \phi; \Delta; (R[r_d : \mathit{int}(x*y)], S) \vdash I\end{array}}{\phi; \Delta; (R, S) \vdash \mathtt{mul}\ r_d, r_s, v; I} \ \textbf{(type-mul)}$$

$$\frac{\begin{array}{c}\phi; \Delta; (R, S) \vdash r_s : \mathit{int}(x) \quad \phi; \Delta; (R, S) \vdash v : \mathit{int}(y) \\ \phi \models y \neq 0 \quad \phi; \Delta; (R[r_d : \mathit{int}(x/y)], S) \vdash I\end{array}}{\phi; \Delta; (R, S) \vdash \mathtt{div}\ r_d, r_s, v; I} \ \textbf{(type-div)}$$

Figure 9: The typing rules for DTAL(I)

$$\frac{\phi;\Delta;(R,S)\vdash r_s : prod(\tau_1,\ldots,\tau_n) \quad \phi\models 0\le i<n \quad \phi;\Delta;(R[r_d:\tau_i],S)\vdash I}{\phi;\Delta;(R,S)\vdash \texttt{load}\, r_d, r_s(i); I}\ \textbf{(type-load-tuple)}$$

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

$$\frac{\phi;\Delta;\Sigma\vdash r_d : \tau\ array(x) \quad \phi;\Delta;\Sigma\vdash v : int(y) \quad \phi\models 0\le y<x \quad \phi;\Delta;\Sigma\vdash r_s <: \tau \quad \phi;\Delta;\Sigma\vdash I}{\phi;\Delta;\Sigma\vdash \texttt{store}\, r_d(v), r_s; I}\ \textbf{(type-store-array)}$$

$$\frac{\phi;\Delta;(R[r:\tau],S)\vdash I}{\phi;\Delta;(R,\tau::S)\vdash \texttt{pop}\, r; I}\ \textbf{(type-pop)}$$

$$\frac{\phi;\Delta;(R,S)\vdash v : \tau \quad \phi;\Delta;(R,\tau::S)\vdash I}{\phi;\Delta;(R,S)\vdash \texttt{push}\, v; I}\ \textbf{(type-push)}$$

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

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

$$\frac{\phi;\Delta;\Sigma\vdash r : int(x) \quad \phi,x=0;\Delta;\Sigma\vdash I \quad \phi;\Delta;\Sigma\vdash v : state(\lambda\Delta'.\lambda\phi'.\Sigma') \quad \phi,x\neq 0\vdash\theta:\phi' \quad \phi,x\neq 0;\Delta\vdash\Theta:\Delta' \quad \phi,x\neq 0;\Delta;\Sigma\models_c \Sigma'[\Theta][\theta]}{\phi;\Delta;\Sigma\vdash \texttt{bne}\, r, v; I}\ \textbf{(type-bne)}$$

$$\frac{}{\phi;\Delta;\Sigma\vdash \texttt{halt}}\ \textbf{(type-halt)}$$

Figure 10: The typing rules for DTAL(II)

Judgment form	Judgement meaning
$\phi \models P$	The proposition P holds under the context ϕ in the integer domain.
$\phi; \Delta \models \tau_1 \equiv \tau_2$	The types τ_1 and τ_2 are equivalent under the context $\phi; \Delta$ modulo constraint satisfaction.
$\phi; \Delta \models \tau_1 \leq \tau_2$	The type τ_1 coerces into the type τ_2 under the context $\phi; \Delta$ modulo constraint satisfaction.
$\phi; \Delta; \Sigma \models_e R$	r_i is of type $R(i)$ under the context $\phi; \Delta; \Sigma$ for every $i \in \mathbf{dom}(R)$ modulo constraint satisfaction.
$\phi; \Delta; \Sigma \models_e S$	Given $\Sigma = (R, \tau_1 :: \ldots :: \tau_n :: \beta)$, S must be of form $(\tau'_1 :: \ldots :: \tau'_n :: \beta)$ and $\phi; \Delta; \Sigma \models \tau_i \equiv \tau'_i$ is derivable for every $1 \leq i \leq n$.
$\phi; \Delta; \Sigma \models_e (R, S)$	This means both $\phi; \Delta; \Sigma \models_e R$ and $\phi; \Delta; \Sigma \models_e S$ are derivable.
$\phi; \Delta; \Sigma \models_c R$	The type of r_i under the context $\phi; \Delta; \Sigma$ coerces into $R(i)$ for every $i \in \mathbf{dom}(R)$ modulo constraint satisfaction.
$\phi; \Delta; \Sigma \models_c S$	Given $\Sigma = (R, \tau_1 :: \ldots :: \tau_n :: \beta)$, S must be of form $(\tau'_1 :: \ldots :: \tau'_n :: \beta)$ and $\phi; \Delta; \Sigma \models \tau_i \leq \tau'_i$ is derivable for every $1 \leq i \leq n$.
$\phi; \Delta; \Sigma \models_c (R, S)$	This means both $\phi; \Delta; \Sigma \models_c R$ and $\phi; \Delta; \Sigma \models_c S$ are derivable.
$\phi; \Delta; \Sigma \vdash_\Lambda v : \tau$	The value v is of type τ under the context $\phi; \Delta; \Sigma$ and the label map Λ.
$\phi; \Delta; \Sigma \vdash_\Lambda v <: \tau$	The value v is of some type τ_1 under the context $\phi; \Delta; \Sigma$ and τ_1 coerces into τ.
$\phi; \Delta; \Sigma \vdash I$	The instruction sequence I is typable.

Figure 11: A summary for various forms of judgments

which can be derived as follows, where Λ is the label mapping of P.

$$\frac{\vdash_\Lambda B_1[\text{well-typed}] \quad \cdots \quad \vdash_\Lambda B_n[\text{well-typed}]}{\vdash P[\text{well-typed}]} \ \textbf{(type-program)}$$

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

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

We have so far introduced various forms of judgments, some of which have yet to be defined later. In Figure 11, we summarize the meaning of these judgments informally.

3 Type Equality and Coercion

As we have mentioned before, a novelty in DML is the separation between language expressions and type index objects. 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 this moment how a register can be used as a type index object since it is mutable. The separation allows us to simply avoid such a problematic issue. There is another advantage, that is, the separation enables us to choose a relatively simple domain for type index objects so that constraints (on type index objects) 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 rules for type equality and coercion, which exhibit clearly the involvement of constraints in type-checking.

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

We present the syntax for constraints as follows.

$$\begin{array}{rrcl} \text{index constraints} & \Phi & ::= & \top \mid P \mid P \supset \Phi \mid \forall a:\gamma.\Phi \\ \text{satisfiability relation} & & & \phi \models P \end{array}$$

The satisfiability relation $\phi \models P$ means that the formula $(\phi)P$ is satisfiable in the domain of integers, where $(\phi)P$ is defined below.

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

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

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

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 $\phi;\Delta$. The rules for type equality and coercion are presented in Figure 12 and Figure 13, respectively. There are some obvious restrictions on some of these rules. For instance, we require that index variable a in the premise of the rule **(type-eq-exi-ivar)** have no free occurrences in the conclusion of this rule.

Notice that we *cannot* replace the rule **(coerce-array)** with the following one.

$$\frac{\phi;\Delta \models \tau \leq \tau' \quad \phi \models x = y}{\phi;\Delta \models \tau\ array(x) \leq \tau'\ array(y)}\ \textbf{(coerce-array}'\textbf{)}$$

This rule can readily make the type system unsound as demonstrated in the following example. Note that `[r0: int(0) array(2)]` stands for a state type $state(\lambda(\rho).\Sigma)$, where $\Sigma = (R, \rho)$ and $R(0) = int(0)\ array$ and $R(i) = unit$ for all other $i \in \mathbf{dom}(R)$.

```
start: [r0: int(0) array(2)]
       mov    r1, r0
       jmp    next
next:  [r0: int(0) array, r1: int array(2)]
       store   r1(0), 1
       ...
```

If the above "seemingly natural" rule is allowed, we can coerce type $int(0)\ array(2)$ into type $int\ array(2)$. Therefore, we can type the above instruction sequence. Notice that r_0 points to a pair of value $(1,0)$ on heap after the store instruction is executed, but the type of r_0 is still $int(0)\ array(2)$. This leads to the unsoundness of the system. In summary, array types are not covariant in DTAL. The plain reason is that arrays are mutable data structure allocated on heap and pointers to arrays may be shared.

Notice that product types in DTAL are covariant. The reason is that a tuple on heap is not mutable in DTAL. If we intend to support tuples in which some components are mutable,

$$\frac{\alpha \in \Delta}{\phi; \Delta \models \alpha \equiv \alpha} \textbf{ (type-eq-tvar)}$$

$$\frac{\phi \models x = y}{\phi; \Delta \models \mathit{int}(x) \equiv \mathit{int}(y)} \textbf{ (type-eq-int)}$$

$$\frac{\phi; \Delta \models \tau_1 \equiv \tau_2 \quad \phi \models x = y}{\phi; \Delta \models \tau_1\ \mathit{array}(x) \equiv \tau_2\ \mathit{array}(y)} \textbf{ (type-eq-array)}$$

$$\frac{\phi; \Delta \models \tau_1 \equiv \tau_1' \quad \cdots \quad \phi; \Delta \models \tau_n \equiv \tau_n'}{\phi; \Delta \models \mathit{prod}(\tau_1, \ldots, \tau_n) \equiv \mathit{prod}(\tau_1', \ldots, \tau_n')} \textbf{ (type-eq-prod)}$$

$$\frac{\phi, a : \gamma; \Delta \models \tau_1 \equiv \tau_2}{\phi; \Delta \models \exists a : \gamma.\tau_1 \equiv \exists a : \gamma.\tau_2} \textbf{ (type-eq-exi-ivar)}$$

$$\frac{\phi; \Delta, \alpha \models \tau_1 \equiv \tau_2}{\phi; \Delta \models \exists \alpha.\tau_1 \equiv \exists \alpha.\tau_2} \textbf{ (type-eq-exi-tvar)}$$

$$\frac{\phi, \phi'; \Delta, \Delta'; \Sigma_1 \models_e \Sigma_2 \quad \phi, \phi'; \Delta, \Delta'; \Sigma_2 \models_e \Sigma_1}{\phi; \Delta \models \mathit{state}(\lambda\Delta'.\lambda\phi'.\Sigma_1) \equiv \mathit{state}(\lambda\Delta'.\lambda\phi'.\Sigma_2)} \textbf{ (type-eq-state)}$$

$$\frac{\phi; \Delta; \Sigma \models_e R \quad \phi; \Delta; \Sigma \models_e S}{\phi; \Delta; \Sigma \models_e (R, S)} \textbf{ (type-eq-reg-stack)}$$

$$\frac{\phi; \Delta; \Sigma \vdash r_i : \tau_i \quad \phi; \Delta \models \tau_i \equiv R(i)}{\phi; \Delta; \Sigma \models_e R} \textbf{ (type-eq-reg)}$$

$$\frac{}{\phi; \Delta; (R, []) \models_e []} \textbf{ (type-eq-stack-empty)}$$

$$\frac{}{\phi; \Delta; (R, \beta) \models_e \beta} \textbf{ (type-eq-stack-var)}$$

$$\frac{\phi; \Delta \models \tau \equiv \tau' \quad \phi; \Delta; \Sigma \models_e S}{\phi; \Delta; (R, \tau :: S) \models_e \tau' :: S'} \textbf{ (type-eq-stack)}$$

Figure 12: Type equality rules for DTAL

$$\frac{\phi; \Delta \vdash \tau : *}{\phi; \Delta \models \tau \leq \mathit{top}} \textbf{ (coerce-top)}$$

$$\frac{}{\phi; \Delta \models \mathit{unit} \leq \mathit{unit}} \textbf{ (coerce-unit)}$$

$$\frac{\alpha \in \Delta}{\phi; \Delta \models \alpha \leq \alpha} \textbf{ (coerce-type-var)}$$

$$\frac{\phi \models x = y}{\phi; \Delta \models \mathit{int}(x) \leq \mathit{int}(y)} \textbf{ (coerce-int)}$$

$$\frac{\phi; \Delta \models \tau_1 \equiv \tau_2 \quad \phi \models x = y}{\phi; \Delta \models \tau_1\ \mathit{array}(x) \leq \tau_2\ \mathit{array}(y)} \textbf{ (coerce-array)}$$

$$\frac{\phi; \Delta \models \tau_1 \leq \tau'_0 \quad \cdots \quad \phi; \Delta \models \tau_n \leq \tau'_{n-1}}{\phi; \Delta \models \mathit{prod}(\tau_0, \ldots, \tau_{n-1}) \leq \mathit{prod}(\tau'_0, \ldots, \tau'_{n-1})} \textbf{ (coerce-prod)}$$

$$\frac{\phi, a : \gamma; \Delta \models \tau_1 \leq \tau_2}{\phi; \Delta \models \exists a : \gamma.\tau_1 \leq \tau_2} \textbf{ (coerce-exi-ivar-l)}$$

$$\frac{\phi \vdash x : \gamma \quad \phi; \Delta \models \tau_1 \leq \tau_2\{a := x\}}{\phi; \Delta \models \tau_1 \leq \exists a : \gamma.\tau_2} \textbf{ (coerce-exi-ivar-r)}$$

$$\frac{\phi; \Delta, \alpha \models \tau_1 \leq \tau_2}{\phi; \Delta \models \exists \alpha.\tau_1 \leq \tau_2} \textbf{ (coerce-exi-tvar-l)}$$

$$\frac{\phi; \Delta, \alpha \models \tau_1 \leq \tau_2\{\alpha := \tau\}}{\phi; \Delta \models \tau_1 \leq \exists \alpha.\tau_2} \textbf{ (coerce-exi-tvar-r)}$$

$$\frac{\phi, \phi_2 \vdash \theta : \phi_1 \quad \phi, \phi_2; \Delta, \Delta_2 \vdash \Theta : \Delta_1 \quad \phi, \phi_2; \Delta, \Delta_2; \Sigma_2 \models_c \Sigma_1[\Theta][\theta]}{\phi; \Delta \models \mathit{state}(\lambda\Delta_1.\lambda\phi_1.\Sigma_1) \leq \mathit{state}(\lambda\Delta_2.\lambda\phi_2.\Sigma_2)} \textbf{ (coerce-state)}$$

$$\frac{\phi; \Delta; \Sigma \models_c R \quad \phi; \Delta; \Sigma \models_c S}{\phi; \Delta; \Sigma \models_c (R, S)} \textbf{ (coerce-reg-stack)}$$

$$\frac{\phi; \Delta; \Sigma \vdash r_i : \tau_i \quad \phi; \Delta \models \tau_i \leq R(i)}{\phi; \Delta; \Sigma \models_c R} \textbf{ (coerce-reg)}$$

$$\frac{}{\phi; \Delta; (R, []) \models_c []} \textbf{ (coerce-stack-empty)}$$

$$\frac{}{\phi; \Delta; (R, \beta) \models_c \beta} \textbf{ (coerce-stack-var)}$$

$$\frac{\phi; \Delta \models \tau \leq \tau' \quad \phi; \Delta; (R, S) \models_c S'}{\phi; \Delta; (R, \tau :: S) \models_c \tau' :: S'} \textbf{ (coerce-stack)}$$

Figure 13: Type coercion rules for DTAL

we must modify the rule **(coerce-prod)**. For instance, if we have a type constructor $prod_2$ for forming types of pair whose first component is mutable but the second is not, then we need the following coercion rule.

$$\frac{\phi, \Delta \models \tau_0 \equiv \tau_0' \quad \phi, \Delta \models \tau_1 \leq \tau_1'}{\phi, \Delta \models prod_2(\tau_0, \tau_1) \leq prod_2(\tau_0', \tau_1')}$$

The need for type coercion is immediate. Foremost, we need type coercion to type jumps as demonstrated in the rules **(type-jmp)**, **(type-beq)** and **(type-bne)**. Also we need type coercion to type the following code sequence since we must show that the type $int(0) * int(0)$ coerces into the type $\exists a : nat.int(a) * int(a)$.

```
start: [r0: int(0) * int(0)]
       jmp    next
next:  [r0: {n:nat} (int(n) * int(n))]
       ...
```

The most noticeable coercion rule is **(coerce-state)**. Informally speaking, a state type is "stronger" if a state is "weaker". The intuitive explanation is that a state type roughly represents the notion of code continuation. We will elaborate on this point when we establish the soundness of the type system of DTAL in Section 5.

We define substitutions on both index and type variables as follows.

$$\begin{array}{rrcl} \text{index variable substitutions} & \theta & ::= & [] \mid \theta[a \mapsto i] \\ \text{type variable substitutions} & \Theta & ::= & [] \mid \Theta[\rho \mapsto S] \mid \Theta[\alpha \mapsto \tau] \end{array}$$

We omit the details on how substitution is performed, which is standard. Given a term $\bullet$ such as a type or a state, we use $\bullet[\Theta]$ ($\bullet[\theta]$) for the result from applying Θ (θ) to $\bullet$. We introduce two forms of judgments $\phi \vdash \theta : \phi'$ and $\phi; \Delta \vdash \Theta : \Delta'$ and present as follows the rules for deriving such judgments.

$$\frac{}{\phi \vdash [] : \cdot}\ \textbf{(subst-iempty)} \qquad \frac{\phi \vdash \theta : \phi' \quad \phi \vdash i : \gamma}{\phi \vdash \theta[a \mapsto i] : \phi', a : \gamma}\ \textbf{(subst-ivar)}$$

$$\frac{\phi \vdash \theta : \phi' \quad \phi \models P[\theta]}{\phi \vdash \theta : \phi', P}\ \textbf{(subst-prop)}$$

$$\frac{}{\phi \vdash [] : \cdot_{\mathrm{tv}}}\ \textbf{(subst-tempty)} \qquad \frac{\phi; \Delta \vdash \Theta : \Delta' \quad \phi; \Delta \vdash S[\text{well-formed}]}{\phi; \Delta \vdash \Theta[\rho \mapsto S] : \Delta', \rho}\ \textbf{(subst-svar)}$$

$$\frac{\phi; \Delta \vdash \Theta : \Delta' \quad \phi; \Delta \vdash \tau : *}{\phi \vdash \Theta[\alpha \mapsto \tau] : \Delta', \alpha}\ \textbf{(subst-tvar)}$$

The following lemmas establish some expected properties on both type equality and coercion.

Lemma 3.1 *We have the following.*

1. *If $\phi; \Delta \vdash \tau : *$ is derivable, then $\phi; \Delta \models \tau \equiv \tau$ is also derivable.*
2. *If $\phi; \Delta \models \tau_1 \equiv \tau_2$ is derivable, then $\phi; \Delta \models \tau_2 \equiv \tau_1$ is also derivable.*
3. *If both $\phi; \Delta \models \tau_1 \equiv \tau_2$ and $\phi; \Delta \models \tau_2 \equiv \tau_3$ are derivable, then $\phi; \Delta \models \tau_1 \equiv \tau_3$ is also derivable.*

Proof This follows an inspection of the rules in Figure 12. ■

```
subscript: ('r,'a){s: nat, i: int}
           [sp: int(i) :: 'a array(s) :: [sp: 'a :: 'r] :: 'r]
           pop        r0           // r0 <- sp[1]
           blt        r0, ERROR    // i is negative
           pop        r1           // r1 <- sp[1]
           arraysize  r2, r1       // obtain the array size
           sub        r2, r0, r2   // r2 <- r0 - r2
           blte       r2, ERROR    // if i >= s
           load       r0, r1(r0)   // r0 <- r1(r0): this is a safe load
           pop        r1           // r1 <- sp[1]
           push       r0           // push r0 onto the stack
           jmp        r1           // done
```

Figure 14: DTAL for array subscript function

Lemma 3.2 *We have the following.*

1. *If $\phi; \Delta \models \tau_1 \equiv \tau_2$ is derivable, then $\phi; \Delta \models \tau_1 \leq \tau_2$ is also derivable.*
2. *If both $\phi; \Delta \models \tau_1 \leq \tau_2$ and $\phi; \Delta \models \tau_2 \leq \tau_3$ are derivable, then $\phi; \Delta \models \tau_1 \leq \tau_3$ is also derivable.*

Proof This follows an inspection of the rules in Figure 12 and Figure 13. ■

Lemma 3.3 *Assume that both $\phi \vdash \theta : \phi'$ and $\phi; \Delta \vdash \Theta : \Delta'$ are derivable.*

1. *If $\phi'; \Delta' \vdash \tau_1 \equiv \tau_2$ is derivable, then $\phi; \Delta \vdash \tau_1[\Theta][\theta] \equiv \tau_2[\Theta][\theta]$ is also derivable.*
2. *If $\phi'; \Delta' \vdash \tau_1 \leq \tau_2$ is derivable, then $\phi; \Delta \vdash \tau_1[\Theta][\theta] \leq \tau_2[\Theta][\theta]$ is also derivable.*

Proof By structural induction on the derivations of $\phi'; \Delta' \vdash \tau_1 \equiv \tau_2$ and $\phi'; \Delta' \vdash \tau_1 \leq \tau_2$, respectively. ■

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

4 Examples

It is simply too overwhelming to formally explain how type-checking in DTAL is performed through even a tiny example because of the involvedness of the type system. Instead, we present in an informal manner how to type-check the DTAL code in Figure 14 so as to facilitate comprehension.

This code sketches an implementation of array subscript function in which run-time array bound checks are performed. We use

```
('r,'a){s:nat,i:int}[sp: int(i) :: 'a array(s) :: [sp: 'a :: 'r] :: 'r]
```

for $state(\lambda(\rho, \alpha).\lambda(s : nat, i : int).\Sigma)$, where

$$\Sigma = (R, int(i) :: (\alpha)array(s) :: \sigma :: \rho)$$

No.	ϕ	Δ	Σ
01	$i : int$	ρ, α	$([], int(i) :: (\alpha) array(s) :: \sigma :: \rho)$
02	$i : int$	ρ, α	$([r_0 : int(i)], (\alpha) array(s) :: \sigma :: \rho)$
03	$i : int, i \geq 0$	ρ, α	$([r_0 : int(i)], (\alpha) array(s) :: \sigma :: \rho)$
04	$i : int, i \geq 0$	ρ, α	$([r_0 : int(i), r_1 : (\alpha) array(s)], \sigma :: \rho)$
05	$i : int, i \geq 0$	ρ, α	$([r_0 : int(i), r_1 : (\alpha) array(s), r_2 : int(s)], \sigma :: \rho)$
06	$i : int, i \geq 0$	ρ, α	$([r_0 : int(i), r_1 : (\alpha) array(s), r_2 : int(i-s)], \sigma :: \rho)$
07	$i : int, i \geq 0, i - s < 0$	ρ, α	$([r_0 : int(i), r_1 : (\alpha) array(s), r_2 : int(i-s)], \sigma :: \rho)$
08	$i : int, i \geq 0, i - s < 0$	ρ, α	$([r_0 : \alpha, r_1 : (\alpha) array(s), r_2 : int(i-s)], \sigma :: \rho)$
09	$i : int, i \geq 0, i - s < 0$	ρ, α	$([r_0 : \alpha, r_1 : \sigma, r_2 : int(i-s)], \rho)$
10	$i : int, i \geq 0, i - s < 0$	ρ, α	$([r_0 : \alpha, r_1 : \sigma, r_2 : int(i-s)], \alpha :: \rho)$

Figure 15: Contexts $\phi_i; \Delta_i; \Sigma_i$ for $i = 1, \ldots, 10$

such that $R(i) = unit$ for all $i \in \mathbf{dom}(R)$, and σ is the state type $state(\Sigma_1)$ such that $\Sigma_1 = state(R, \alpha :: \rho)$.

Notice that numbers are inserted into the code so that we can readily identify each instruction in the code. We use the label `ERROR` for the entry to some code reporting bound violations. Also we use `arraysize r2, r1` for an instruction which stores in `r2` the size of the array to which `r1` points. Intuitively speaking, when the code execution reaches the label `subscript`, `sp` points to a stack whose top three cells store an integer, a pointer to some array and a label, respectively. The type of the label states that the `sp` must be of type $\alpha :: \rho$ when the execution jumps to the label. The type system of DTAL guarantees that the part of stack that is typed by ρ can neither be read nor be written during the subsequent code execution. The simple reason is that neither `pop` nor `push`, the only two instructions involving stack, can be applied when the type of a stack is a stack type variable.

Let ins_i be the ith instruction and I_i be $ins_i; \ldots; ins_{10};$`halt` for $1 \leq i \leq 10$. We argue that there is a derivation $\mathcal{D}$ with the following conclusion.

$$i : int; \alpha; (R, int(i) :: (\alpha) array(s) :: \sigma :: \rho) \vdash I_1$$

Then we need to derive derivations $\mathcal{D}_i$ with conclusions of form $\phi_i; \Delta_i; \Sigma_i \vdash I_i$ for $i = 1, \ldots, 10$. We list these contexts $\phi_i; \Delta_i; \Sigma_i$ in Figure 15. Note that $\Sigma_{10} = (R, \alpha :: \rho)$ for some R. Therefore,

$$\phi_{10}; \Delta_{10}; \Sigma_{10} \models_c \alpha :: \rho$$

is derivable. This implies that $\phi_{10}; \Delta_{10}; \Sigma_{10} \vdash I_{10}$ is derivable. It is straightforward to verify that $\phi_i; \Delta_i; \Sigma_i \vdash I_i$ are derivable for $i = 1, \ldots, 9$. Notice that we need to prove that $\phi_7 \models 0 \leq i < s$ when deriving $\phi_7; \Delta_7; \Sigma_7 \vdash I_7$, but this is trivial since $i \geq 0$ and $i - s < 0$ are assume in the context ϕ_7.

We now present a more sophisticated example. In Figure 16, the Xanadu program implements a binary search function on an integer array. The type system of Xanadu guarantees that this implementation is memory safe and it is unnecessary to perform array bounds checking at run-time. The syntax following the keyword `invariant` is basically a state type stating that there exist integers i and j satisfying $0 \leq i \leq n$ and $0 \leq j + 1 \leq n$ such that variables `low` and `high` have types $int(i)$ and $int(j)$, respectively, at this program point. This is treated as a loop invariant for the `while` loop that follows. Some further explanation can be found in (Xi 1999a). The DTAL code in Figure 17 (loosely) corresponds to the Xanadu implementation of the binary

```
{n:nat} int bsearch(key: int, vec: <int> array(n)) {

  var:
    int low, mid, high, x;;

  low = 0;
  high = arraysize(vec) - 1;

  invariant:
    [i:int, j:int | 0 <= i <= n, 0 <= j+1 <= n] (low: int(i), high: int(j))
  while (low <= high) {
    mid = (low + high) / 2;
    x = vec[mid];
    if (key == x) { return mid; }
    else if (key < x) { high = mid - 1; }
         else { low = mid + 1; }
  }

  return -1;
}
```

Figure 16: An implementation of binary search in Xanadu

search. It can be verified that the DTAL code is also memory safe. We give an intuitive but informal explanation as follows.

When the code execution reaches the label `loop`, the integers i and j are stored in `r2` and `r3` such that

$$0 \leq i \leq n \quad \text{and} \quad 0 \leq j+1 \leq n,$$

where n is the size of the array to which `r0` points. It can be readily inferred that the integer in `r4` equals $\lfloor (i+j)/2 \rfloor$ and $i \leq j$ holds when the load instruction is executed. Clearly, we have the following

$$0 \leq i \leq \lfloor (i+j)/2 \rfloor \leq j \leq n-1,$$

and therefore the load instruction is memory safe. This guarantees the memory safety of the code since the load instruction is the only memory operation in the code.

Notice that the types attached to the labels in this example seem intractable to synthesize in practice. This supports the view that, in order to generate memory safety proofs for large programs, it is necessary to have a high level source language such as Xanadu in which the programmer can supply type annotations. We are currently investigating how to compile these annotations into a low level language such as DTAL.

5 Soundness

We recall that a machine state $\mathcal{M}$ is a triple $(\mathcal{H}, \mathcal{R}, \mathcal{S})$, where $\mathcal{H}$ and $\mathcal{R}$ are finite mappings representing heap and register file, respectively, and $\mathcal{S}$ is a list which stands stack. Given a heap address h, $\mathcal{H}(h)$ is a tuple $(hc_0, \ldots, hc_{n-1})$ such that every hc_i is either a constant or a heap address. We use a judgment of form $\mathcal{H} \models_\Lambda hc : \tau$ to mean that the heap address or constant hc

```
bsearch:  {n: nat} [r0: int array(n), r1: int(n), r5: int]
                            // r5 stores the key value we are trying to find
          mov     r2, 0      // r2 stores the lower bound
          sub     r3, r1, 1  // r3 stores the upper bound
          jmp     loop

loop:     {n: nat, i: int, j: int |
           0 <= i <= n /\ 0 <= j+1 <= n}  // this is the loop invariant
          [r0: int array(n), r2: int(i), r3: int(j), r5: int]
          sub     r6, r2, r3
          bgt     r6, notfound         // if r2 > r3
          add     r4, r2, r3           // r4 <- (r2 + r3)
          div     r4, r4, 2            // r4 <- (r2 + r3)/2
          load    r1, r0(r4)           // r1 <- r0(r4)
          sub     r6, r5, r1
          blt     r6, less             // if r5 < r1
          bgt     r6, greater          // if r5 > r1
          mov     r31, r4              // r5 = r1: key found
          jmp     finish

less:     {n: nat, i: int, j: int, k: nat |
           0 <= i <= n /\ 0 <= j+1 <= n /\ i <= j /\ k = (i+j) / 2 }
          [r0: int array(n), r2: int(i), r3: int(j), r4: int(k), r5: int]
          sub     r3, r4, 1
          jmp     loop

greater:  {n: nat, i: int, j: int, k: nat |
           0 <= i <= n /\ 0 <= j+1 <= n /\ i <= j /\ k = (i+j) / 2 }
          [r0: int array(n), r2: int(i), r3: int(j), r4: int(k), r5: int]
          add     r2, r4, 1
          jmp     loop

notfound: []
          mov   r31, -1
          jmp   finish

finish:   [r31: int] // r31 contains the index of the key or -1(not found)
          halt       // the program halts
```

Figure 17: DTAL code for binary search on an integer array

$$\frac{}{\mathcal{H} \models_\Lambda i : int(i)} \textbf{ (heap-int)} \qquad \frac{\cdot; \cdot_{\text{tv}} \vdash \Lambda(l) \leq \sigma}{\mathcal{H} \models_\Lambda l : \sigma} \textbf{ (heap-label)}$$

$$\frac{}{\mathcal{H} \models_\Lambda hc : unit} \textbf{ (heap-unit)}$$

$$\frac{\mathcal{H} \models_\Lambda hc : \tau\{a := i\} \quad \cdot \vdash i : \gamma}{\mathcal{H} \models_\Lambda hc : \exists a : \gamma.\tau} \textbf{ (heap-exists)}$$

$$\frac{\mathcal{H}(h) = (hc_0, \ldots, hc_{n-1}) \quad \mathcal{H} \models_\Lambda hc_0 : \tau_0 \quad \cdots \quad \mathcal{H} \models_\Lambda hc_{n-1} : \tau_{n-1}}{\mathcal{H} \models_\Lambda h : prod(\tau_0, \ldots, \tau_{n-1})} \textbf{ (heap-prod)}$$

$$\frac{\mathcal{H}(h) = (hc_0, \ldots, hc_{n-1}) \quad \mathcal{H} \models_\Lambda hc_0 : \tau \quad \cdots \quad \mathcal{H} \models_\Lambda hc_{n-1} : \tau}{\mathcal{H} \models_\Lambda h : (\tau) array(n)} \textbf{ (heap-array)}$$

$$\frac{\mathcal{H} \models_\Lambda \mathcal{R}(i) : R(i) \quad \text{for all } 0 \leq i < n_r}{(\mathcal{H}, \mathcal{R}) \models_\Lambda R} \textbf{ (heap-register)}$$

$$\frac{}{(\mathcal{H}, []) \models_\Lambda []} \textbf{ (heap-stack-empty)}$$

$$\frac{\mathcal{H} \models_\Lambda hc : \tau \quad (\mathcal{H}, \mathcal{S}) \models_\Lambda S}{(\mathcal{H}, hc :: \mathcal{S}) \models_\Lambda \tau :: S} \textbf{ (heap-stack)}$$

$$\frac{(\mathcal{H}, \mathcal{R}) \models_\Lambda R \quad (\mathcal{H}, \mathcal{S}) \models_\Lambda S}{(\mathcal{H}, \mathcal{R}, \mathcal{S}) \models_\Lambda (R, S)} \textbf{ (heap-state)}$$

Figure 18: Rules for modeling states

is of type τ under the heap $\mathcal{H}$ and the label map Λ. For $\mathcal{M} = (\mathcal{H}, \mathcal{R}, \mathcal{S})$ and $\Sigma = (R, S)$, the judgment $\mathcal{M} \models \Sigma$ means that the machine state $\mathcal{M}$ entails or models the state Σ.

We use the judgment $\mathcal{M} \models \phi; \Delta; \Sigma$ to mean the context $\phi; \Delta; \Sigma$ is satisfied under the machine state $\mathcal{M}$, which can be derived with the following rule.

$$\frac{\cdot \vdash \theta : \phi \quad \cdot; \cdot_{\text{tv}} \vdash \Theta : \Delta \quad \mathcal{M} \models \Sigma[\Theta][\theta]}{\mathcal{M} \models \phi; \Delta; \Sigma}$$

Lemma 5.1 *(Substitution) If $\phi; \Delta; \Sigma \vdash I$, $\cdot \vdash \theta : \phi$ and $\cdot; \cdot_{\text{tv}} \vdash \Theta : \Delta$ is derivable, then $\cdot; \cdot_{\text{tv}}; \Sigma[\Theta][\theta] \vdash I$ is also derivable.*

Proof This follows from a careful inspection of type equality rules, type coercion rules and typing types for DTAL. ■

Lemma 5.2 *Let $\mathcal{M} = (\mathcal{H}, \mathcal{R}, \mathcal{S})$. Assume $\mathcal{H} \models hc : \tau_1$ is derivable. If $\cdot; \cdot_{\text{tv}} \models \tau_1 \leq \tau_2$ is derivable, then $\mathcal{H} \models hc : \tau_2$ is also derivable.*

Proof We proceed by a structural induction on the derivation $\mathcal{D}_1$ of $\mathcal{H} \models hc : \tau_1$ and the derivation $\mathcal{D}$ of $\cdot; \cdot_{\text{tv}} \models \tau_1 \leq \tau_2$. If τ_1 is a state type σ, then $\mathcal{D}_1$ is of the following form.

$$\frac{\cdot; \cdot_{\text{tv}} \vdash \Lambda(l) \leq \sigma}{\mathcal{H} \models_\Lambda l : \sigma} \textbf{ (heap-label)}$$

By Lemma 3.2, $\cdot; \cdot_{\text{tv}} \models \Lambda(l) \leq \tau_2$ is derivable. Thus $\mathcal{H} \models_\Lambda l : \tau_2$ is derivable. We now assume that τ_1 is not a state type and present some interesting cases.

$$\boxed{\mathcal{D}_2 = \dfrac{\cdot;\cdot_{\text{tv}} \models \tau_1 \equiv \tau_2 \quad \cdot \models x = y}{\cdot;\cdot_{\text{tv}} \models \tau_1\ \mathit{array}(x) \leq \tau_2\ \mathit{array}(y)}}$$

Since $\mathcal{H} \models hc : \tau_1\ \mathit{array}(x)$ is derivable, the derivation $\mathcal{D}_1$ is of the following form, where $n = x = y$.

$$\frac{\mathcal{H}(h) = (hc_0, \ldots, hc_{n-1}) \quad \mathcal{H} \models_\Lambda hc_0 : \tau_1 \quad \cdots \quad \mathcal{H} \models_\Lambda hc_{n-1} : \tau_1}{\mathcal{H} \models_\Lambda hc : \tau_1\ \mathit{array}(n)}\ \textbf{(heap-array)}$$

By Lemma 3.2 (1), we can derive $\cdot;\cdot_{\text{tv}} \models \tau_1 \leq \tau_2$. By induction hypothesis, we can derive $\mathcal{H} \models_\Lambda hc_i : \tau_2$ for $0 \leq i < n$. This leads to the following, and we conclude the case.

$$\frac{\mathcal{H}(h) = (hc_0, \ldots, hc_{n-1}) \quad \mathcal{H} \models_\Lambda hc_0 : \tau_2 \quad \cdots \quad \mathcal{H} \models_\Lambda hc_{n-1} : \tau_2}{\mathcal{H} \models_\Lambda hc : \tau_2\ \mathit{array}(n)}\ \textbf{(heap-array)}$$

$$\boxed{\mathcal{D}_2 = \dfrac{a : \gamma; \cdot_{\text{tv}} \models \tau \leq \tau_2}{\cdot;\cdot_{\text{tv}} \models \exists a : \gamma.\tau \leq \tau_2}}$$

Note that $\tau_1 = \exists a : \gamma.\tau$. Thus, $\mathcal{D}_1$ is of the following form.

$$\frac{\mathcal{H} \models_\Lambda hc : \tau\{a := i\} \quad \cdot \vdash i : \gamma}{\mathcal{H} \models_\Lambda hc : \exists a : \gamma.\tau}\ \textbf{(heap-exists)}$$

By Lemma 3.3 (1), we can derive $\cdot;\cdot_{\text{tv}} \vdash \tau\{a := i\} \leq \tau_2$. By induction hypothesis on (1), $\mathcal{H} \models hc : \tau_2$ is derivable.

$$\boxed{\mathcal{D}_2 = \dfrac{\cdot;\cdot_{\text{tv}} \models \tau \leq \tau\{a := i\} \quad \cdot \vdash i : \gamma}{\cdot;\cdot_{\text{tv}} \models \tau_1 \leq \exists a : \gamma.\tau}}$$

Note that $\tau_2 = \exists a : \gamma.\tau$. Then $\mathcal{H} \models hc : \tau\{a := i\}$ is derivable by induction hypothesis on (1), and this leads to the following.

$$\frac{\mathcal{H} \models_\Lambda hc : \tau\{a := i\} \quad \cdot \vdash i : \gamma}{\mathcal{H} \models_\Lambda hc : \exists a : \gamma.\tau}\ \textbf{(heap-exists)}$$

The rest of the cases can be treated similarly. ■

Lemma 5.3 *Assume $\mathcal{M} \models \phi_1; \Delta_1; \Sigma_1$ is derivable. If $\phi_1; \Delta_1; \Sigma_1 \models_c \Sigma_2[\Theta][\theta]$ is derivable for some θ and Θ such that both $\phi_1 \vdash \theta : \phi_2$ and $\phi_1; \Delta_1 \vdash \Theta : \Delta_2$ are also derivable, then $\mathcal{M} \models \phi_2 : \Delta_2; \Sigma_2$ is derivable.*

Proof The lemma follows from a straightforward application of Lemma 5.2 to the derivation of $\phi_1; \Delta_1; \Sigma_1 \models_c \Sigma_2[\Theta][\theta]$.

Notice that the derivation of $\mathcal{M} \models \phi_1; \Delta_1; \Sigma_1$ must be of the following form.

$$\frac{\cdot \vdash \theta_1 : \phi_1 \quad \cdot;\cdot_{\text{tv}} \vdash \Theta_1 : \Delta_1 \quad \mathcal{M} \models \Sigma_1[\Theta_1][\theta_1]}{\mathcal{M} \models \phi_1; \Delta_1; \Sigma_1}$$

Let $(R, S) = \Sigma_1[\theta_1][\Theta_1]$. By Lemma 3.3, we can find θ' and Θ' such that $\cdot \vdash \theta' : \phi_2$, $\cdot;\cdot_{\text{tv}} \vdash \Theta' : \Delta_2$ and $\cdot;\cdot_{\text{tv}}; (R, S) \models \Sigma_2[\Theta'][\theta']$ are derivable. Let $(R', S') = \Sigma_2[\Theta'][\theta']$.

Note that $(\mathcal{H}, \mathcal{R}) \models R$ is derivable since $\mathcal{M} \models \Sigma$ is derivable. In other words, we can derive $\mathcal{H} \models \mathcal{R}(i) : R(i)$ for $0 \leq i < n_r$. Also notice $\cdot;\cdot_{\text{tv}} \models R(i) \leq R'(i)$ are derivable for $0 \leq i < n_r$ since $\cdot;\cdot_{\text{tv}}; (R, S) \models_c (R', S')$ is derivable. By Lemma 5.2, $\mathcal{H} \models \mathcal{R}(i) : R'(i)$ for

$0 \leq i < n_r$. Hence, $(\mathcal{H}, \mathcal{R}) \models R'$ is derivable. Similarly, we can derive $(\mathcal{H}, \mathcal{S}) \models S'$. A derivation of $\mathcal{M} \models (R', S')$ is thus obtained as follows.

$$\frac{(\mathcal{H}, \mathcal{R}) \models_\Lambda R' \quad (\mathcal{H}, \mathcal{S}) \models_\Lambda S'}{(\mathcal{H}, \mathcal{R}, \mathcal{S}) \models_\Lambda \Sigma'} \textbf{ (heap-state)}$$

This yields a derivation of $\mathcal{M} \models \phi_2; \Delta_2; \Sigma_2$. ■

Assume that $\mathcal{D}$ is a derivation of $\vdash B$[well-typed] for a block $B = \lambda\Delta.\lambda\phi.(\Sigma, I)$, where I is a list of instructions $ins_0, \ldots, ins_{n-1}$. We use $I(i)$ for ins_i and $I[i]$ for $ins_i, \ldots, ins_{n-1}$, that is, the suffix of I beginning at $I(i)$. Then there are the *greatest* subderivations $\mathcal{D}(i)$ of $\mathcal{D}$ for $0 \leq i < n$ with conclusions of form $\phi_i; \Delta_i; \Sigma_i \vdash I_i$. Notice that for a given i, there may exists several subderivations of $\mathcal{D}$ with a conclusion where the righthand side of $\vdash$ is I_i because the application of **(type-open-reg)** does not alter the righthand side of $\vdash$. This is the reason why we need the word *greatest* in the above definition.

Assume that $\mathcal{D}$ is a derivation of $\vdash P$[well-typed] for a program $P = (l_1 : B_1, \ldots, l_n : B_n)$, where $B_i = \lambda\Delta_i\lambda\phi_i.(\Sigma_i, I_i)$, and $\mathcal{D}_i$ are the derivations of $\vdash B_i$[well-typed] for $i = 1, \ldots, n$. Let $\Lambda = \Lambda(P)$ and $J = J(P)$. Notice that for every $0 \leq ic < \mathit{length}(J)$, if $J(ic)$ is not a label, then it is an instruction from some block B_k, that is, it is $I_k(i)$ for some k and i. We write $\mathcal{D}(ic)$ for $\mathcal{D}_k(i)$, $J(ic)$ for $I_k(i)$ and $J[ic]$ for $I_k[i]$ in the following presentation.

Lemma 5.4 *Let $\mathcal{M}$ be a machine state. If $\mathcal{M} \models \phi, a : \gamma; \Delta; (R[r : \tau], S)$ is derivable, then $\mathcal{M} \models \phi; \Delta; (R[r : \exists\alpha.\tau], S)$ is also derivable.*

Proof This follow from a structural induction on the derivation of

$$\mathcal{M} \models \phi, a : \gamma; \Delta; (R[r : \tau], S)$$

■

In order to establish the soundness of the type system of DTAL, we need to prove that $\mathcal{M}' \models_\Lambda \phi'; \Delta'; \Sigma'$ holds if $\mathcal{M} \models_\Lambda \phi; \Delta; \Sigma$ and $(ic, \mathcal{M}) \to (ic', \mathcal{M}')$ are derivable, where $\mathcal{D}(ic)$ and $\mathcal{D}(ic')$ are $\phi; \Delta; \Sigma \vdash J[ic]$ and $\phi'; \Delta'; \Sigma' \vdash J[ic']$, respectively. In other words, we should justify the typing rules in Figure 9 and Figure 10 with respect to the effects on machine states resulted from the execution of instructions. Unfortunately, this cannot succeed unless we impose some regularity condition on the derivation of $\mathcal{M} \models_\Lambda \mathcal{D}(ic)$. We present the main reason for this as follows.

Suppose that there are two pointers stored in registers r_1 and r_2 which point to the same address on heap in which integer 0 is stored. It is possible that r_1 and r_2 at this moment have type $(\mathit{int})\mathit{array}(1)$ and $(\mathit{int}(0))\mathit{array}(1)$, respectively. If we update the address with integer 1 through the pointer in r_1, the update cannot be seen by r_2. Therefore, the update leads to inconsistency since the type of r_2 is still $(\mathit{int}(0))\mathit{array}(1)$. A more formal description can be given as follows. Let $\Sigma = (R, S)$ be a state, where we have $R(1) = (\mathit{int})\mathit{array}(1)$, $R(2) = (\mathit{int}(0))\mathit{array}(1)$ and $S = []$. Also let $\mathcal{M} = (\mathcal{H}, \mathcal{R}, \mathcal{S})$ such that $\mathcal{H}(h) = 0$ for some heap address and $\mathcal{R}(1) = \mathcal{R}(2) = h$ and $\mathcal{S}$ is empty. It can be readily verified that $\mathcal{M} \models \Sigma$ is derivable. According to the typing rules, We are now allowed to execute the instruction `store r1(0), 1` since the type of `r1` is $(\mathit{int})\mathit{array}(1)$. This changes the $\mathcal{M}$ into $\mathcal{M}' = (\mathcal{H}', \mathcal{R}, \mathcal{S})$ where $\mathcal{H}'(h) = 1$. Obviously, we cannot derive $\mathcal{H}' \models r_2 : (\mathit{int}(0))\mathit{array}(1)$ (we actually have $\mathcal{H}' \models r_2 : (\mathit{int}(1))\mathit{array}(1)$), and therefore, $\mathcal{M}' \models \Sigma$ does not hold.

Definition 5.5 *(Regularity) Let $\mathcal{H}$ be a heap mapping. For every $h \in \mathbf{dom}(\mathcal{H})$ such that $\mathcal{H}(h)$ is a tuple $(hc_0, \ldots, hc_{n-1})$, we use $h[i]$ to represent the heap address in which hc_i is stored, where i ranges over $0, \ldots, n-1$.*

Let $\mathcal{T}$ be a partial mapping from heap addresses to closed types and $\mathcal{D}$ be a derivation of $\mathcal{M} \models \Sigma$. If for all applications of the following rules in $\mathcal{D}$,

$$\frac{\mathcal{H}(h) = (hc_0, \ldots, hc_{n-1}) \quad \mathcal{H} \models_\Lambda hc_0 : \tau \quad \cdots \quad \mathcal{H} \models_\Lambda hc_{n-1} : \tau}{\mathcal{H} \models_\Lambda h : (\tau)\mathit{array}(n)} \ \textbf{(heap-array)}$$

$\cdot; \cdot_{\mathrm{tv}} \models \tau \equiv \mathcal{T}(h[i])$ is derivable for every $0 \leq i < n$, then $\mathcal{D}$ is $\mathcal{T}$-regular. We say $\mathcal{D}$ is regular if $\mathcal{D}$ is $\mathcal{T}$-regular for some $\mathcal{T}$. We call $\mathcal{T}$ a regularity mapping.

Let us now argue that the above $\mathcal{M} \models R$ cannot have a regular derivation. Note that we must have a derivation of the following form in order for $\mathcal{H} \models r_1 : (\mathit{int})\ \mathit{array}(1)$ to be derivable, where i is some integer.

$$\frac{\mathcal{H}(h) = (i) \quad \mathcal{H} \models i : \mathit{int}}{\mathcal{H} \models h : \mathit{int}\ \mathit{array}(1)}$$

Similarly, we must have a derivation of the following form for deriving $\mathcal{H} \models r_2 : (\mathit{int}(0))\ \mathit{array}(1)$.

$$\frac{\mathcal{H}(h) = (i) \quad \mathcal{H} \models i : \mathit{int}(0)}{\mathcal{H} \models h : \mathit{int}(0)\ \mathit{array}(1)}$$

We cannot find a regularity mapping $\mathcal{T}$ such that both $\cdot; \cdot_{\mathrm{tv}} \models \mathit{int} \equiv T(h[1])$ and $\cdot; \cdot_{\mathrm{tv}} \models \mathit{int}(0) \equiv T(h[1])$ are derivable since this would imply a derivation of $\cdot; \cdot_{\mathrm{tv}} \models \mathit{int} \equiv \mathit{int}(0)$, which is clearly impossible.

We can now inspect the proof of Lemma 5.3 and observe that the derivation of $\mathcal{M} \models \phi_2 : \Delta_2; \Sigma_2$ is also $\mathcal{T}$-regular if the given derivation of $\mathcal{M} \models \phi_1; \Delta_1; \Sigma_1$ is $\mathcal{T}$-regular.

Lemma 5.6 *(Main Lemma) Assume that P is a program. Let $\Lambda = \Lambda(P)$ and $J = J(P)$ and $\mathcal{D}$ be a derivation of $\vdash P[\text{well-typed}]$. Also let ic be an instruction count such that $\mathcal{D}(ic)$ is a derivation of $\phi; \Delta; \Sigma \vdash J[ic]$. We have the following.*

1. *If $\mathcal{M} \models \phi; \Delta; \Sigma$ is derivable for some machine state $\mathcal{M}$, then $(\mathcal{M}, ic) \rightarrow_P (\mathcal{M}', ic')$ is derivable for some $\mathcal{M}'$ and ic', or $(\mathcal{M}, ic) \rightarrow \mathtt{HALT}$ is derivable.*

2. *If $(ic, \mathcal{M}) \rightarrow_P (ic', \mathcal{M}')$ is derivable and $\mathcal{D}(ic')$ is a derivation of $\phi'; \Delta'; \Sigma' \vdash J[ic']$, then $\mathcal{M}' \models \phi'; \Delta'; \Sigma'$ has a regular derivation if the derivation of $\mathcal{M} \models \phi; \Delta; \Sigma$ is regular.*

Proof The proof follows from an inspection of the typing rules in Figure 9 and Figure 10 and the evaluation rules in Figure 6. By Lemma 5.4, we can assume that the last applied rule in $\mathcal{D}(ic')$ is neither **(type-open-reg)** nor **(type-open-stack)** when we prove (2). Let $\mathcal{M} = (\mathcal{H}, \mathcal{R}, \mathcal{S})$, and we present a few cases below.

- The derivation $\mathcal{D}(ic)$ is of the following form.

$$\frac{\begin{array}{c}\phi; \Delta; (R, S) \vdash r_s : \mathit{int}(x) \quad \phi; \Delta; (R, S) \vdash v : \mathit{int}(y) \\ \phi \models y \neq 0 \quad \phi; \Delta; (R[r_d : \mathit{int}(x/y)], S) \vdash I\end{array}}{\phi; \Delta; (R, S) \vdash \mathtt{div}\ r_d, r_s, v; I} \ \textbf{(type-div)}$$

Since $\mathcal{M} \models \phi; \Delta; \Sigma$ is available, $\mathcal{M}(r_s) = i$ and $\mathcal{M}(v) = j$ for some integers i and j, and $j \neq 0$ holds. With the evaluation rule **(eval-div)**, we have the following.

$$(ic, (\mathcal{H}, \mathcal{R}, \mathcal{S})) \rightarrow_P (ic + 1, (\mathcal{H}, \mathcal{R}[r_d = i/j], \mathcal{S}))$$

Clearly, $(\mathcal{H}, \mathcal{R}[r_d = i/j], \mathcal{S}) \models \phi; \Delta; (R[r_d = x/y], S)$ is derivable.

- The derivation $\mathcal{D}(ic)$ is of the following form, where the applied rule is **(type-beq)**.

$$\frac{\begin{array}{ccc} \phi; \Delta; \Sigma \vdash r : int(x) & \phi, x \neq 0; \Delta; \Sigma \vdash I & \phi; \Delta; \Sigma \vdash v : state(\lambda\Delta'\lambda\phi'.\Sigma') \\ \phi, x = 0 \vdash \theta : \phi' & \phi, x = 0; \Delta \vdash \Theta : \Delta' & \phi, x = 0; \Delta; \Sigma \models_c \Sigma'[\Theta][\theta] \end{array}}{\phi; \Delta; \Sigma \vdash \texttt{beq}\, r, v; I}$$

Since $\mathcal{M} \models \phi; \Delta; \Sigma$ is available, $\mathcal{R}[r_d]$ is some integer i. We now have two cases.

  - $i \neq 0$ holds. The evaluation rule **(eval-beq-false)** yields the following.

$$(ic, (\mathcal{H}, \mathcal{R}, \mathcal{S})) \rightarrow_P (ic + 1, (\mathcal{H}, \mathcal{R}, \mathcal{S}))$$

    It is trivial that $(\mathcal{H}, \mathcal{R}, \mathcal{S}) \models \phi, x \neq 0; \Delta; \Sigma$ is derivable.

  - $i = 0$ holds. Note that we know $\mathcal{M}(v)$ is some label l since $\mathcal{M} \models v : \sigma$ is derivable for some state type σ. This means $J^{-1}(l)$ is well-defined. Let ic' be $J^{-1}(l)$, we derive the following with the evaluation rule **(eval-beq-true)** when $i = 0$ holds.

$$(ic, (\mathcal{H}, \mathcal{R}, \mathcal{S})) \rightarrow_P (ic' + 1, (\mathcal{H}, \mathcal{R}, \mathcal{S}))$$

    Note that $\mathcal{M} \models l : state(\lambda\Delta'.\lambda\phi'.\Sigma')$. This implies that $\Lambda(l) = \sigma$ for some state type σ such that $\cdot; \cdot_{\mathrm{tv}} \vdash \sigma \leq state(\lambda\Delta'.\lambda\phi'.\Sigma')$. Assume $\sigma = state(\lambda\Delta''.\lambda\phi''.\Sigma'')$. Then the following is derivable for some θ and Θ such that both $\phi' \vdash \theta : \phi''$ and $\phi'; \Delta' \vdash \Theta : \Delta''$ are derivable.

$$\phi'; \Delta'; \Sigma' \models_c \Sigma''[\Theta][\theta]$$

    By Lemma 5.3, we can readily derive $(\mathcal{H}, \mathcal{R}, \mathcal{S}) \models \phi''; \Delta''; \Sigma''$.

- The derivation $\mathcal{D}(ic)$ is of the following form.

$$\frac{\begin{array}{ccc} \phi; \Delta; \Sigma \vdash r_d : \tau\ array(x) & \phi; \Delta; \Sigma \vdash v : int(y) & \\ \phi \models 0 \leq y < x & \phi; \Delta; \Sigma \vdash r_s <: \tau & \phi; \Delta; \Sigma \vdash I \end{array}}{\phi; \Delta; \Sigma \vdash \texttt{store}\, r_d(v), r_s; I}\ \textbf{(type-store-array)}$$

Since $\mathcal{M} \models \phi; \Delta; \Sigma$ is available, there exist n and i such that $0 \leq i < n$, $\mathcal{M}(v) = i$, $\mathcal{M}(r_d) = h$ for some heap address h and $\mathcal{H}(h) = (hc_0, \ldots, hc_{n-1})$ for some $hc_0, \ldots, hc_{n-1}$. Therefore, we have the following by the evaluation rule **(eval-store)**, where $hc = \mathcal{M}(r_s)$.

$$(ic, (\mathcal{H}, \mathcal{R}, \mathcal{S})) \rightarrow (ic + 1, (\mathcal{H}[h \mapsto (hc_0, \ldots, hc_{i-1}, hc, hc_{i+1}, \ldots, hc_{n-1})], \mathcal{R}, \mathcal{S}))$$

The derivation of $(\mathcal{H}[h \mapsto (hc_0, \ldots, hc_{i-1}, hc, hc_{i+1}, \ldots, hc_{n-1})], \mathcal{R}, \mathcal{S}) \models \phi; \Delta; \Sigma$ is straightforward as it can be readily proven that $\mathcal{M} \models hc : \tau$ (this is the place where we need the regularity condition).

- The derivation $\mathcal{D}(ic)$ is of the following form.

$$\frac{\phi; \Delta; (R[r:\tau], S) \vdash I}{\phi; \Delta; (R, \tau :: S) \vdash \texttt{pop}\ r; I}\ \textbf{(type-pop)}$$

  Since $\mathcal{M} \models \phi; \Delta; \Sigma$ is available for $\Sigma = (R, \tau :: S)$, $\mathcal{S}$ must be of form $hc_0 :: \ldots :: hc_{n-1} :: []$ for some $n \geq 1$. With the evaluation rule **(eval-pop)**, we have the following.

$$(ic, (\mathcal{H}, \mathcal{R}, hc_0 :: \ldots :: hc_{n-1} :: [])) \rightarrow (ic+1, (\mathcal{H}, \mathcal{R}[r = hc_0], hc_1 :: \ldots :: hc_{n-1} :: []))$$

  The derivation of $(\mathcal{H}, \mathcal{R}[r = hc_0], hc_1 :: \ldots :: hc_{n-1} :: []) \models \phi; \Delta; (R[r:\tau], S)$ is straightforward.

- $\mathcal{D}(ic)$ is of the following form.

$$\frac{}{\phi; \Delta; \Sigma \vdash \texttt{halt}}\ \textbf{(type-halt)}$$

  Obviously, we can derive $(ic, (\mathcal{H}, \mathcal{R}, \mathcal{S})) \rightarrow_P \texttt{HALT}$ with the evaluation rules **(eval-halt)**.

The rest of the cases can be handled in a similar manner. ■

Theorem 5.7 *(Progress) Let $P = (l_1 : B; \ldots; l_n : B_n)$ be a program and $\Lambda = \Lambda(P)$ and $J = J(P)$. Assume $\vdash P$[well-typed] is derivable and $\Lambda(l_1) = \exists\rho(R_{empty}, \rho)$. If $(0, \mathcal{M}_0) \rightarrow^* (ic, \mathcal{M})$ then either $(ic, \mathcal{M}) \rightarrow \texttt{HALT}$, or $(ic, \mathcal{M}) \rightarrow (ic', \mathcal{M}')$ for some ic' and $\mathcal{M}'$. In other words, the execution of a well-typed program in DTAL either halts normally or runs forever.*

Proof Let $\mathcal{D}$ be the derivation of P. Then $\mathcal{D}(0)$ is a derivation of $\cdot; \rho; (R_{empty}, \rho) \vdash J[0]$. Clearly, there is a regular derivation of $\mathcal{M}_0 \models \cdot; \rho; (R_{empty}, \rho)$. The theorem then follows from Lemma 5.6. ■

6 Extension with Sum Types

The programmer can declare in Xanadu a polymorphic union type as in Figure 19 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` (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. For instance, we informally explain the meaning of the `switch` statement in Figure 19; 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. 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 this 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. Unfortunately, it cannot be inferred in the type system of TAL that a tag which does not indicate `Nil` must indicate `Cons` in this case, and this makes it difficult in TAL to handle some optimization in pattern compilation.[2] In general, the type system of TAL contains some limitations on handling sum or union types.

[2]Though it is possible in TAL to use some macros for handling the case where there are only two constructors, it seems difficult to handle a general case involving more than two constructors.

```
('a) union list =
  { Nil; 'a * <'a> list Cons }

('a) int length (xs: <'a> list) {
  var: int x;;
  x = 0;
  while (true) {
    switch(xs) {
      case Nil: return x;
      case Cons(_, xs): x = x + 1;
    }
  }
}
```

Figure 19: A length function on lists in Xanadu

$$\frac{\phi;\Delta \models \tau_i \equiv \tau \quad \phi \models x = i}{\phi;\Delta \models \mathit{choose}(x,\tau_0,\ldots,\tau_{n-1}) \equiv \tau}\ \textbf{(type-eq-choose-l)}$$

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

$$\frac{\phi \models x = y \quad \phi;\Delta \models \tau_0 \equiv \tau'_0 \quad \cdots \quad \phi;\Delta \models \tau_{n-1} \equiv \tau'_{n-1}}{\phi;\Delta \models \mathit{choose}(x,\tau_0,\ldots,\tau_{n-1}) \equiv \mathit{choose}(y,\tau'_0,\ldots,\tau'_{n-1})}\ \textbf{(type-eq-choose)}$$

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

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

Figure 20: Additional type equality and coercion rules

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 $\mathit{sum}(\tau_0,\ldots,\tau_{n-1})$, 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 $\mathit{choose}(x,\tau_0,\ldots,\tau_{n-1})$ to stand for a type which must be one of $\tau_0,\ldots,\tau_{n-1}$, determined by the value of x: the type is τ_i if $x = i$. Also we present some additional rules in Figure 20 for handling type coercion involving sum types (rules for type equality are omitted). Note nat_n is the sort $\{a : \mathit{int} \mid 0 \leq a < n\}$ for every natural number n.

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

$$\exists a : \mathit{nat}_n.\mathit{int}(a) * \mathit{choose}(a,\tau_0,\ldots,\tau_{n-1}),$$

that is, a value of type $\mathit{sum}(\tau_0,\ldots,\tau_{n-1})$ is represented as a pair in which the first part is a tag determining the type of the second part. We need the following typing rule for *choose*.

$$\frac{\phi, a = 0;\Delta;\Sigma[r : \tau_1] \vdash I \quad \cdots \quad \phi, a = n-1;\Delta;\Sigma[r : \tau_{n-1}] \vdash I}{\phi;\Delta;\Sigma[r : \mathit{choose}(a,\tau_0,\ldots,\tau_{n-1})] \vdash I}\ \textbf{(type-choose)}$$

```
length: ('r, 'a) [sp: 'a list :: [sp: int :: 'r] :: 'r]
        // [sp: int :: '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
        // and the program would not type-check in TAL
        pop     r1          // store the list argument into r1
        mov     r2, 0       // initialize r2

loop:   ('r, 'a) [r2: int, r1: 'a list, sp: [sp: int :: 'r] :: 'r]
        unfold  r1          // r1: unit + 'a * 'a list
        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 (r3 = 1 since r3 is not 0)
        load    r1, r1(1)   // move list tail into r1
        add     r2, r2, 1   // increase r2 by 1
        jmp     loop        // loop again

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

Figure 21: An implementation of the length function on lists in DTAL

We now present an example to illustrate the use of sum types. The usual list *type constructor* can be represented as $\Lambda\alpha.\mu t.unit + \alpha * t$, where Λ abstracts over types and μ is the fixed point operator on types. Note that we use t for a type variable here. As usual, the following rules are needed for handling μ operator.

$$\frac{\phi; \Delta \vdash v : \tau\{t := \mu t.\tau\}}{\phi; \Delta \vdash v : \mu t.\tau} \textbf{ (type-fold)}$$
$$\frac{\phi; \Delta \vdash v : \mu t.\tau}{\phi; \Delta \vdash v : \tau\{t := \mu t.\tau\}} \textbf{ (type-unfold)}$$

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 21 corresponds to the Xanadu program in Figure 19. The state type following the label `length` indicates that the top element on the stack 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. Note that the code would not type-check if translated into TAL.

The DTAL code in Figure 22 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. For instance, we can introduce

```
<int> list upto(int n) {
   var: <int> list xs;;

   xs = Nil;
   while (n >= 0) {
     xs = Cons(n, xs); n = n - 1;
   }
   return xs;
}

upto:   ('r){i:int} [sp:int(i) :: 'r]
        pop                     r1
        push                    <>
        push                    0
        newtuple[int list]      r2 // r2 <- []
        jmp                     loop

loop:   ('r){i:int} [r1:int(i), r2: int list, sp: 'r]
        blt                      r1, finish
        push                     r2
        push                     r1
        newtuple[int * int list] r2 // r2 <- (r1, r2)
        push                     r2
        push                     1
        newtuple[int list]       r2 // r2: int list
        sub                      r1, r1, 1
        jmp                      loop

finish: [r2: int list]
        halt
```

Figure 22: Implementations of the upto function in Standard ML and DTAL

$$\frac{\phi;\Delta;\Sigma \vdash r : \tau \quad \Delta \models \|\tau\| \not\equiv \mathit{unit} \quad \phi;\Delta;\Sigma \vdash I}{\phi;\Delta;\Sigma \vdash \mathtt{bnu}\ r, v; I} \ \textbf{(type-bnu-false)}$$

$$\frac{\begin{array}{c}\phi;\Delta;\Sigma \vdash v : \mathit{state}(\lambda\Delta'.\lambda\phi'.\Sigma') \\ \phi \vdash \theta : \phi' \qquad \phi \vdash \Theta : \Delta' \qquad \phi;\Delta;\Sigma \models_c \Sigma'[\Theta][\theta]\end{array}}{\phi;\Delta;\Sigma \vdash \mathtt{bnu}\ r, v; I} \ \textbf{(type-bnu-true)}$$

Figure 23: Typing rules for the instruction `bnu`

```
upto:    ('r){i:nat} [sp: int(i) :: 'r]
         pop                      r1
         mov                      r2, <>
         jmp                      loop

loop:    ('r){i:nat} [r1: int(i), r2: int list, sp: 'r]
         blt                      r1, finish
         push                     r2
         push                     r1
         newtuple[int * int list] r2 // r2 <- (r1, r2)
         sub                      r1, r1, 1
         jmp                      loop

finish: [r2: int list]
         halt
```

Figure 24: Another Implementation of the upto function in DTAL

the branch instruction `bnu r, v` , which branches to the label in v if the value in r equals $\langle\rangle$. The typing rules for `bnu` are listed in Figure 23. This leads to the more concise DTAL code in Figure 24.

The example in Figure 25 is adopted from (Necula and Lee 1996), which is clearly more involved. Given a list in which every element is either an integer or a pair of integers, the following code sum up all the integers in such a list. For instance, if the given list is $(1,2) :: 3 :: []$, then the answer is $1+2+3 = 6$. We declare a datatype `single_or_pair` to make this a homogeneous list in ML. We use the optimized list representation in the DTAL code.

The treatment of sum types extends the one in (Harper and Stone 1998). 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 $\mathit{int}(i-1) * \mathit{choose}(i-1, \tau_1, \tau_2)$ and the subtyping relation can be derived with the use of type coercion rules.

```
union single_or_pair = {
  int Single; int * int Pair
}

int sum (l: <single_or_pair> list) {
  var:
    int s, i, i1, i2;
    single_or_pair x;;

  s = 0;
  while (true) {
    switch (l) {
      case Nil: return s;
      case Cons(x, l):
        switch (x) {
          case Single (i): s = i + s;
          case Pair(i1, i2): s = i1 + i2 + s;
        }
    }
  }
  return s;
}
```

```
start:  [r0: (int + (int * int)) list]
        mov    r1, 0

loop:   [r0: (int + (int * int)) list, r1: int]
        bnu      finish    // r0 is empty
        load     r2, r0(0) // r2: int + (int * int)
        load     r3, r2(0) // load tag into r3
        bne      r3, pair  //
        load     r2, r2(1) // r2: sum(0, int, int * int)
        add      r1, r1, r2
        load     r0, r0(1) // load the tail: r0: (int + (int * int)) list
        jmp      loop

pair:   [r0: (int + (int * int)) list, r1: int, r2: sum(1, int, int * int)]
        load     r2, r2(1) // r2: sum(1, int, int * int)
        load     r3, r2(0) // r2: int * int
        add      r1, r1, r3
        load     r3, r2(1) // r2: int * int
        add      r1, r1, r3
        load     r0, r0(1) // load the tail into r0
        jmp      loop

finish: [r1: int]
        halt
```

Figure 25: Tallying up numbers appearing in a list

7 Implementation

7.1 Type-checker for DTAL

We have prototyped a type-checker and an interpreter for DTAL and verified many examples. The implementation and examples are available on-line (Xi 1999b). There is certain amount of non-determinism in the the typing rules for DTAL. In the implementation, we impose some restriction to eliminate the non-determinism. For instance, when both of the rules **(coerce-exi-ivar-l)** and **(coerce-exi-ivar-r)** are applicable, we choose the former over the latter. For the rule **(type-open-reg)**, we currently apply it whenever it is applicable. An alternative is provide an auxiliary instruction `open` r to indicate the need of an application of this rule to register r.

Notice that we have not explained how to obtain the type index expressions $\vec{x}$ in the premise of the rule **(type-jmp)**. In practice, if we impose certain syntactic restriction on forming state types, these type index expressions can always be inferred through unification.[3] This subject is studied in Chapter 4 (Xi 1998). This strategy is adopted in the current implementation. From the point of view of type-checking, it also seems reasonable to require that a DTAL program be annotated with these type index expressions. For instance, we can use the following form of instruction

$$\texttt{jmp}\ v[x_1, \ldots, x_n]$$

to indicate that $\vec{x} = x_1, \ldots, x_n$ are the type index expressions needed for typing `jmp` v as is presented in the rule **(type-jmp)**. The conditional branching instructions can be given a similar form.

We currently only accept linear constraints on integers and solve them with a method based on Fourier-Motzkin variable elimination (Dantzig and Eaves 1973). Though the linear integer programming problem is NP-complete in general, the typical constraints generated from type-checking DTAL code are simple and can be effectively solved. Relevant experience can be found in (Xi and Pfenning 1998). We have verified many DTAL examples (including all the ones in this paper) with the type-checker, some of which are available on-line (Xi 1999b).

7.2 Compilation into DTAL

We briefly mention 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 (Necula and Lee 1998) and Popcorn (Morrisett et al. 1999). The most significant feature of Xanadu is its type system, which supports a restricted form of dependent types. Please see (Xi 1999a) for details. The compilation is essentially like compiling C into a typical untyped assembly language except that we need to construct state types for labels this time. We have compiled all the examples in this paper.[4]

We currently do not perform register allocation when compiling Xanadu into DTAL and execute the generated DTAL code with an interpreter. The typability of DTAL code is unaffected by register allocation and spill. The argument is the same as the one for arguing that the certifiability of proof-carrying code is unaffected by register allocation and spill (Necula 1998).

[3]The restriction is that for every state type $\mathit{state}(\lambda\Delta.\lambda\phi.\Sigma)$, every variable declared in ϕ should be used at least once as a single type index expression to index a type in Σ.

[4]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 while leaving the raw versions at (Xi 1999b).

7.2.1 Synthesis

One approach to generating DTAL code is to synthesize state types for labels. For the function `copy` in Figure 1, we map variables `src`, `dst`, `length` and `i` to `r1`, `r2`, `r3` and `r4` respectively, and readily generate the code in Figure 2 excluding the state types for labels `copy`, `loop` and `finish` (this is *exactly* like a compilation from C into a typical assembly language).

We briefly explain how to form these state types. The state type for `copy` is a directly translation from the type of the function `copy`. We synthesize the state type for `loop` with some informal reasoning. When the execution first reaches the label `loop`, we know that for some natural numbers m and n satisfying $m \leq n$ the types of `r1`, `r2`, `r3` and `r4` are $(int)\,array(m), (int)\,array(n), int(m)$ and $int(0)$, respectively. It can be readily verified by analyzing the loop body that the values in `r1`, `r2` and `r3` stay the same and the integer value in `r4` can only increase. Since the initial value in `r4` is 0, the value in `r4` is always a natural number during the execution of the loop. This yields the state type for `loop`. The state type for `finish` is trivial.

In general, we identify those integer variables in a loop whose values can only increase or decrease during the execution of the loop and name them monotonic variables. Suppose that the initial value of a monotonic variable x is i_0 and x is mapped to register r. We then assume in the state type attached to the loop that r is of type $int(i)$ for some integer $i \geq i_0$ or $i \leq i_0$ according whether x is increasing or decreasing. This is a simple and widely applicable heuristic. Actually, this is the heuristic used in the Touchstone compiler for array bound check elimination. However, it is also clear that this heuristic is too limited to handle other more sophisticated cases such as binary search where there is a non-monotonic array index (this heuristic is even ineffective for the trivial program in Figure 26).

7.2.2 Annotation

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 handled in practice. In Figure 26, 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. Various larger examples, which are too unwieldy to present, can be found at (Xi 1999b). 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 $int(a)$ and $int(b)$, respectively, where a and b are natural numbers satisfying $a + b = n$. Note that n is the size of array `vec`. Xanadu has a sound type system as proven in (Xi 1999a), but we do not have to rely on this fact in this paper. We merely assume that the type annotations in Xanadu are hints to a compiler.

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 26. 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. If we compile a well-type program in Xanadu, we expect that the generated DTAL code is guaranteed to type-check (assuming the compiler is implemented correctly) but this is yet to be rigorously proven. In the case where the source program in Xanadu may not be well-typed, we can always ignore

```
{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 26: Implementations of an initialization functions in Xanadu and DTAL

type annotations and compile with the synthesis approach if the generated DTAL code does not type-check (though we have not experimented with this option).

8 Related Work

DTAL is designed on top of TAL with a dependent type system to overcome some limitations. While inheriting most features from TAL, DTAL also alters some. For instance, a type in TAL can be annotated with a flag to indicate the initialization status of a value with this type, but we adopt a different strategy in DTAL to handle initialization. We simply use *top* to represent the type of all uninitialized values. This strategy works because every tuple (array) is initialized upon allocation. DTAL 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.

The notion of proof-carrying code introduced in (Necula 1997) 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 (Necula and Lee 1998), 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 VC generator (Floyd 1967), generating verification conditions for both type safety and memory safety. TPCC also performs some loop invariant synthesis for eliminating array bound checks. TPCC seems a bit heavy-handed for handling type safety when compared to TAL and it needs to be studied whether TPCC can readily handle higher-order functions since TPCC uses essentially a first-order logic to capture invariants. For instance, it may be desirable to express something like the following: this function call can only be made if the called function takes an integer and returns a natural number. This property, which can be readily expressed with dependent types, seems to require higher-order logic if expressed with predicates.

DML is a functional programming language that enriches ML with a restricted form of dependent types (Xi and Pfenning 1999), allowing the programmer to capture more program invariants through types and thus detect more program errors at compile-time. In particular, the programmer can refine datatypes with type index expressions in DML, capturing more invariants in various data structures. 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 (Xi and Pfenning 1998).

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.

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. This includes establishing the soundness of the type system of DTAL and implementing 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 continue the study on compiling Xanadu into DTAL, which we expect to be mostly straightforward. A similar but more challenging task is to construct a compiler from DML into DTAL. 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 gratefully acknowledge some discussion with Jim Hook on the subject of the paper and thank Chad Brown for proofreading a draft of this paper and providing us with many valuable comments.

References

Dantzig, G. and B. Eaves (1973). Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A) 14*, 288–297.

Floyd, R. W. (1967). Assigning meanings to programs. In J. T. Schwartz (Ed.), *Mathematical Aspects of Computer Science*, Volume 19 of *Proceedings of Symposia in Applied Mathematics*, Providence, Rhode Island, pp. 19–32. American Mathematical Society.

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

Milner, R., M. Tofte, R. W. Harper, and D. MacQueen (1997). *The Definition of Standard ML*. Cambridge, Massachusetts: MIT Press.

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

Morrisett, G., K. Crary, N. Glew, and D. Walker (1998, March). Stack-based typed assembly language. In *Proceedings of Workshop on Types in Compilation*.

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

Necula, G. (1997). Proof-carrying code. In *Conference Record of 24th Annual ACM Symposium on Principles of Programming Languages*, pp. 106–119. ACM press.

Necula, G. (1998, September). *Compiling with Proofs*. Ph. D. thesis, Carnegie Mellon University. Also available as technical report No. CMU-CS-98-154.

Necula, G. and P. Lee (1996). Proof-carrying code. Technical Report CMU-CS-96-165, Carnegie Mellon University.

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

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

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

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

Xi, H. (1999a). Dependent Types in Imperative Programming. Current version at `http://www.cse.ogi.edu/~hongwei/academic/papers/Xanadu.ps`.

Xi, H. (1999b). Implementations and Examples for Xanadu and DTAL. Available at `http://www.cse.ogi.edu/~hongwei/Xanadu-DTAL`.

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

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