TALx86: A Realistic Typed Assembly Language

Greg Morrisett Karl Crary Neal Glew Dan Grossman Richard Samuels
Frederick Smith David Walker Stephanie Weirich Steve Zdancewic
Cornell University

Abstract

In previous work, we presented a formalism for a statically typed, idealized assembly language called TAL. The goal of TAL was to provide an extremely low-level, statically-typed target language that is better suited than Java bytecodes for supporting a wide variety of source languages and a number of important optimizations.

In this paper, we present our progress in defining and implementing a realistic typed assembly language called TALx86. The TALx86 instructions comprise a relatively complete fragment of the Intel IA32 (32-bit 80x86 flat model) assembly language and are thus executable on processors such as the Intel Pentium. The type system for the language incorporates a number of advanced features necessary for safely compiling large programs to good code.

To motivate the design of the type system, we present a type-safe, C-based language called Popcorn and show how various Popcorn features are compiled to TALx86.

1 Introduction

The ability to type-check low-level or object code, such as Java Virtual Machine Language (JVML) bytecodes [10], allows an extensible system to verify the preservation of an important class of safety properties when untrusted code is added to the system. For example, a web browser can check memory safety of applets, ensuring they will not corrupt arbitrary data. Indeed, the entire JDK 1.2 security model depends crucially upon the ability of the JVML type system to prevent untrusted code from by-passing run-time checks that are meant to enforce the high-level security policy.

To support portability and type-checking, the JVML was defined at a relatively high-level of abstraction as a stack-based abstract machine. The language was engineered to make type-checking relatively easy. However, the JVML design suffers from a number of drawbacks:

1. Semantic errors have been uncovered in the JVML verifier and its English specification. Much recent work [1, 16, 18] has concentrated on constructing an *ex post facto* formal model of the language so that a type-soundness theorem can be proven. A by-product of this work is that we now know the design could have been considerably improved had a formal model been constructed in conjunction with the design process.

2. It is difficult (or, at the least, inefficient) to compile high-level languages other than Java to JVML. For instance, approaches for compiling languages with parametric polymorphism have generally involved either code replication [2] or run-time type checks [14]. This has even constrained extensions to Java itself [17]. As another example, definitions of languages such as Scheme [8] dictate that tail calls be implemented in a space-efficient manner. However, the limitations of JVML necessitate that control-flow stacks be explicitly encoded as heap-allocated objects.

3. Although the JVML was designed for ease of interpretation, in practice, just-in-time (JIT) compilers are used to achieve acceptable performance. Since the JIT translation to native code happens *after* verification, an error in the compiler can introduce a security hole. Furthermore, the need for rapid compilation limits the quality of code that a JIT compiler produces.

To address these concerns, we have been studying the design and implementation of type systems for machine languages. The goal of our work is to identify typing abstractions that have general utility for encoding a variety of high-level language constructs and security policies, but that do not interfere with optimization. Such abstractions are necessary even in very expressive contexts such as proof-carrying-code [15].

In previous work [13, 12], we presented a statically typed, RISC-based assembly language called

TAL, showed that a toy functional language could be compiled to TAL, and that the type system for TAL was sound: well-typed assembly programs could not violate the primitive typing abstractions. In later work, we described various extensions to TAL to support stack-allocation of activation records (and other data) [11] and separate type-checking and link-checking of object files [6]. The languages described were extremely simple so as to keep the formalism manageable.

In this paper, we informally describe TALx86, a statically typed variant of the Intel IA32 (32-bit 80x86 flat model) assembly language. The TALx86 type system is considerably more advanced than the simple type systems we have described previously. In addition to providing support for stack-allocation, separate type-checking and linking, and a number of basic type-constructors (*e.g.*, records, tagged unions, arrays, *etc.*), the type system supports higher-order and recursive type-constructors, arbitrary data representation, and a rich kind structure that allows polymorphism for different "kinds" of types.

To demonstrate the utility of these features, we also describe a high-level language called Popcorn and a compiler that maps Popcorn to TALx86. Popcorn is a safe C-based language that provides support for first-class polymorphism, abstract types, tagged unions, exceptions, and a simple module system. Ultimately, Popcorn will support other C-like features such as stack-allocated data and "flattened" data structures.

We begin by giving a brief overview of the process of compiling a Popcorn program to TALx86, verifying the output of the compiler, and creating an executable. We then discuss the salient details of Popcorn. Finally, we present the TALx86 type system by showing how Popcorn programs may be translated to type-correct TALx86 code. We close by discussing planned extensions.

The current software release for TALx86 and Popcorn is available at `http://www.cs.cornell.edu/talc`.

2 TALx86 Tools

This section describes how the TALx86 tools (listed in Table 1) are used together to develop safe native programs. Suppose the Popcorn source for an application is in two files, `foo.pop` and `main.pop`. We compile each file to TALx86 with the commands:

```
% popcorn -c foo.pop
% popcorn -c main.pop
```

If there are no syntax or type errors, then six new files are generated: `foo.tal`, `foo_i.tali`, `foo_e.tali`, `main.tal`, `main_i.tali`, and `main_e.tali`. The `.tal` files are IA32 assembly language with type annotations, as described in Section 4. A `.tal` file also records what values it imports and exports by listing typed *interface* files. Any `extern` declarations are compiled into the corresponding import interface file (`_i.tali`). The types of non-static values are compiled into the corresponding export interface file (`_e.tali`).

To type-check an individual `.tal` file, we use `talc`:

```
% talc -c foo.tal
% talc -c main.tal
```

If the Popcorn compiler is implemented correctly, type-checking the individual `.tal` files that it produces will never fail.

To verify that a collection of files may be safely linked we use the link-verifier:

```
% talc --verify-link foo.tal main.tal
```

Correct Popcorn code may fail to link-check, just as traditional object files may fail to link, due to missing or multiple definitions. However, the link-verifier, unlike a traditional linker, also checks that files agree on the *types* of all shared values. (See Glew and Morrisett [6] for the technical details.)

The `.tal` files can be assembled and linked with traditional tools. They are compatible with MASM (Microsoft's Macro Assembler) except that MASM fails on long lines. We provide an assembler without this deficiency. TALx86 macros are expanded as the file is assembled.

Finally, to produce a stand-alone executable some additional trusted files are linked. One component is the Boehm-Demers-Weiser conservative garbage collector [3] which is responsible for memory management. There is also a small runtime environment that provides essential features such as I/O. Although the runtime cannot be written in TALx86, the types of its values can, so the runtime is revealed to applications via a typed interface file.

We have described the build cycle for an executable in great detail. In practice all of these steps are performed automatically with the single command:

```
% popcorn foo.pop main.pop -o main.exe
```

Although Popcorn is the only "serious" compiler targeting TALx86 at this time, TALx86 is not specifically designed for Popcorn. In fact, we have written a compiler for a small part of Scheme, thus demonstrating the feasibility of compiling a higher-order, dynamically-typed language.

TALx86 tools	
talc	Type-checks a TALx86 file.
link-verifier	Verifies that linking a set of TALx86 files together is safe.
assembler	Assembles a TALx86 file to produce a COFF or ELF object file.
popcorn	Compiles Popcorn to TALx86.
scheme	Compiles a tiny subset of Scheme to TALx86. Written in Popcorn.

Table 1: Components of the TALx86 implementation

3 Popcorn

In this section we informally describe most of Popcorn's features.

Most expressions, statements and declarations in Popcorn are identical to those in C [9]. Unsafe features such as pointer arithmetic, the address operator, and casting, as well as some features such as bit-fields and enumerations, have been omitted. Standard enhancements such as more flexible variable declarations and a C++-like namespace mechanism are also supported.

The remaining differences are mainly due to the type-system. Below we discuss the salient points and give examples.

3.1 Control Flow

The basic control constructs such as `while`, `for`, `do`, `break`, and `continue` are identical to those in C except that test expressions must have type `bool`.[1]

Popcorn's `switch` construct differs from C in that execution never "falls through" cases. Furthermore, a default case is required unless the other cases are exhaustive. The argument of a switch test expression can be an `int`, `char`, `union`, or `exception`. Unions and exceptions are discussed below. For example, we could find the first occurrence of the character `'a'` in an array:

```
int i = 0, answer;
while (true)
  switch arr[i] {
   case 'a': answer = i;
             break; // break from while
   default:  i++;
  }
```

Array subscripts are bounds-checked at runtime (see Section 4.4); the above example will exit immediately if `arr` does not contain an `'a'`.

Exceptions may have different types and exception handlers may switch on the name of an exception, as in Java. However, exception names are not hierarchical.

[1] The result type of relational and logical operators is `bool`.

3.2 Data

Currently, the simple types of Popcorn are `bool`, `int`, `char`, and `string`; we plan to add more (such as `unsigned`) soon. Unlike C, strings are not null-terminated. Arrays carry their size to support bounds-checks. A special `size` construct retrieves the size of an array or string.

Popcorn also has tuples which are useful for encoding anonymous structures and multiple return values. The `new` construct creates a new tuple (as well as new `struct` and `union` values). For example, the following code performs component-wise doubling of a pair of ints:

```
*(int,int) x   = new (3, 4);
*(int,int) dbl = new (x.1+x.1, x.2+x.2);
```

Popcorn has two kinds of structure definitions: `struct` and `?struct`. They resemble `struct *` in C. The difference between `struct` and `?struct` is that values of types defined with `struct` cannot be `null` (a primitive construct in the language). Values of types defined with `?struct` are checked for null on field access; failure causes the program to exit immediately.

Unions in Popcorn are more like ML datatypes than C unions. Each variant consists of a tag and an associated type (possibly void). For example,

```
union tree
{void Leaf; int Numleaf; *(tree,tree)Node};
```

Any value of a `union` type is in a particular variant, as determined by its tag, and may not be treated otherwise. We use `switch` to determine the variant of an expression and bind the corresponding value to a variable. Continuing our example, we can write:

```
int sum(tree e) {
 switch e {
  case Leaf: return 0;
  case Numleaf(x): return x;
```

```
  case Node(x): return sum(x.1)+sum(x.2);
 }
}
```

3.3 Parametric Polymorphism

Popcorn function, `struct`, `?struct`, and `union` declarations may all be parameterized over types. For example, we can define lists as:

```
?struct <'a>list {'a hd; <'a>list tl;}
```

To declare that a variable `x` holds a list of ints, we instantiate the type parameter: `<int>list x`. Explicit type instantiation on expressions is not necessary; for example, `new list(3,null)` has type `<int>list`. Having polymorphic functions means we can write a length function that works on any type of list. Polymorphism is particularly useful with function pointers. For example, we can write a map function:

```
<'b>list map('b f('a), <'a>list l) {
   if (l == null) return null;
   return new list(f(l.hd), map(f, l.tl));
}
```

A call to this function could look like:

```
<int>list x;
...
<string>list y = map(int_to_string, x);
```

4 An overview of TALx86

In this section, we give an overview of the features found in TALx86, and describe via example how those features may be used. In particular, we show how Popcorn code may be compiled to type-correct TALx86.

TALx86 uses the syntax of MASM for instructions and data, and augments it with syntax for type annotations necessary for verification. The type annotations can be broken into the following classes:

1. Import and export interface information – used for separately type-checking object files.

2. Type constructor declarations – used to declare new types and type abbreviations.

3. Typing preconditions on code labels – used to specify the types that registers must have before control may enter the associated code.

4. Types on data labels – used to specify the type of a static data item.

5. Typing coercions on instruction operands – used to coerce values of one type to another.

6. Macro instructions – used to encapsulate small instruction sequences as an atomic action.

The most important of these are the typing preconditions on code labels (3). These annotations are of the general form:

$$\forall \alpha_1{:}\kappa_1 \cdots \alpha_m{:}\kappa_m . \{r_1{:}\tau_1, \cdots, r_n{:}\tau_n\}$$

and are used by the type-checker to ensure that, if control is to be transferred to the corresponding label, then registers r_1 through r_n will contain values of type τ_1 through τ_n respectively. The bound type variables, $\alpha_1, \ldots, \alpha_m$, allow the types on the registers to be polymorphic. One must explicitly instantiate a polymorphic precondition before control can be transferred to the corresponding label. As we will see, TALx86 supports different "kinds" of types. Consequently, each type-variable is explicitly labeled with a kind κ so that we may check that only appropriate types are used to instantiate the bound type variables.

Given a typing precondition for a code label, the type-checker verifies that the instructions in the associated code block are type correct under the assumptions that $\alpha_1, \cdots, \alpha_m$ are *abstract* types, and that r_i has type τ_i. By treating the type variables as abstract types, we are ensured that the code will be type-correct for any appropriate instantiation.

In the rest of this section, we assume that the syntax and semantics of MASM instructions and data will be apparent, and focus our attention on the typing annotations and abstractions. We show how various high-level features from Popcorn may be compiled to TALx86. Due to space limitations, we omit discussion of many TALx86 features, including exceptions, static data, higher-order types, and interfaces.

4.1 Basics

Our first example uses a loop to calculate the sum of the first n natural numbers:

```
int i = n+1;
int s = 0;
while(--i > 0)
  s += i;
```

We could translate the above fragment to the following TALx86 code, assuming n is initially in register `ecx`:

```
    mov     eax,ecx   ; i = n
    inc     eax       ; ++i
    mov     ebx,0     ; s = 0
    jmp     test
body: {eax: B4, ebx: B4}
    add     ebx,eax   ; s += i
test: {eax: B4, ebx: B4}
    dec     eax       ; --i;
    cmp     eax,0     ; i > 0
    jg      body
```

In this example, the label preconditions say the same thing: "control transfer to this code cannot occur unless registers `eax` and `ebx` have `B4` values (4-byte integers) in them." The type-checker uses these constraints to check that the operands to each instruction in each block are safe.

Assume for our example that we know `ecx` initially contains a `B4`. Then after the first instruction, `eax` also has a `B4`. The increment is therefore legal; it is not legal to increment pointers. The third instruction puts a `B4` in `ebx`. Hence the verifier is assured that the precondition for jumping to the test label is satisfied. The test label requires a `B4` in `ebx` even though it does not use the value because it transfers control to `body` which does use it.

Now consider writing a function:

```
int sum(int n) {
  // previous example is the body
  return s;
}
```

Of course, the function must have some way to return to the caller. Assume for the moment that the caller places the return address in register `ebp`. In the code below, the typing precondition assumes that `ecx` contains a 4-byte integer and `ebp` contains a label with its own precondition. In particular, the type annotation `ebp: {eax: B4}` should be read "`ebp` contains a pointer to code that expects a `B4` in `eax`".

```
sum:  {ecx: B4, ebp: {eax: B4}}
    <as above>
body: {eax: B4, ebx: B4, ebp: {eax: B4}}
    <as above>
test: {eax: B4, ebx: B4, ebp: {eax: B4}}
    dec     eax       ; --i;
    cmp     eax,0     ; i > 0
    jg      body      ; if so, goto body
    mov     eax,ebx   ; otherwise,
    jmp     ebp       ; return s
```

The final `jmp` verifies because `eax` contains a `B4`. (Notice it would verify even without the preceding `mov` instruction; type soundness does not guarantee algorithmic correctness.) The type on the `sum` label describes a non-standard calling convention with the argument in `ecx`, the return address in `ebp`, and the result in `eax`. Such a calling convention is typically used for leaf procedures in an optimizing compiler. One way to "call" `sum` is to use `jmp`.

```
    mov     ebp,after
    mov     ecx,10
    jmp     sum
after: {eax: B4}
    <code that uses result>
```

The code explicitly moves the return address (`after`) into `ebp`, moves the integer argument into `ecx`, and then jumps to `sum`. The jump type-checks because the precondition on `sum` requires an integer in `ecx` and a return address in `ebp` that expects an integer in `eax`.

4.2 Stacks and Function Calls

To support richer and more realistic calling conventions, TALx86 has a control-flow stack abstraction and stack types. The following examples demonstrate how these types are used. For a theoretical discussion, refer to Morrisett et al [11].

The standard C calling convention on Win32 requires that the return address be placed on top of the stack,[2] followed by the arguments. Before returning, a function pops the return address. The caller is responsible for popping the arguments.[3]

TALx86 describes the shape of the stack as a list of types, where `se` represents an empty stack and if σ is a stack type, then $\tau::\sigma$ is the type that describes stacks where the top-most element has type τ and the rest of the stack is described by σ. For example,

```
{eax: B4}::B4::B4::se
```

is the type of a stack with three elements: a return address expecting a `B4` in `eax` and then two `B4` values. If a register points to a stack (as `esp` generally does), we write `esp: sptr` σ where σ is a stack type.

If we gave our `sum` function the type, `{esp: sptr {eax: B4}::B4::se}` then we could only call `sum` when the stack contained exactly the return address and the argument. Clearly we would like calls to `sum` to type-check regardless of the depth of the stack. To overcome this problem TALx86 supports *stack polymorphism* to abstract portions of the stack. For example, we may assign `sum` the type:

[2] Stacks "grow" towards lower addresses; the "top" is the lowest address.

[3] Also, `ebp` is callee-save; we will incorporate this shortly.

```
∀ρ:Ts.
{esp: sptr{eax: B4, esp: sptr B4::ρ}::B4::ρ}
```

which says that "For any stack shape ρ, `sum` can be called whenever `esp` contains a pointer to a stack with a suitable return address, followed by an integer, followed by a stack ρ." The code associated with `sum` is type-checked treating ρ as an abstract type.

Notice that if `sum` returns by jumping to the given return address, the stack must have the same shape as on input except without the return address. Indeed, we can assert a much stronger property since `sum` is type-checked holding ρ abstract: The input stack corresponding to ρ will remain unmodified throughout the lifetime of the procedure [4]. Hence, a caller is ensured that `sum` will not read or modify its local data (or that of its caller, *etc.*).

Returning to our example, `mov eax,ecx` at the beginning of `sum` would now become, `mov eax,[esp+4]` so as to load the integer argument from the stack into `eax`. The final `jmp` would be replaced with `retn` which pops the return address and then jumps to it. A call to `sum` must now have an additional annotation which instantiates ρ with the actual stack type (not including the input argument which is not part of ρ). A simple example looks like:

```
main: {esp: se}
    push    42 ; hidden on stack
    push    10 ; input argument
    call    tapp(sum, <B4::se>)
after:
    <code after>
```

where the `call` instruction pushes the return address `after` before jumping, and where the `tapp` instantiates ρ with `B4::se`.

Usually a call will occur in a context where part of the stack is already abstract, so the ρ instantiation will use a stack variable in scope at the call site. Indeed, ρ can be instantiated with a stack type containing ρ! In this respect, TALx86 supports a form of polymorphic recursion. For example, Figure 1 shows a recursive implementation of `sum`. The recursive call says the stack now has one more `B4` and return address on it.

We can also use polymorphism to encode callee-save registers into the calling convention. To force `sum` to preserve the value in `ebp`, we require that `ebp` has a value of distinct abstract type α on entry and exit. We would write:

```
∀α:T4 ρ:Ts.
   {ebp: α, esp: sptr{ebp: α, ...}, ...}
```

where `T4` means that α can be any 4-byte type. A call would now have to instantiate α and ρ appropriately.

TALx86 supports addition of constants to stack pointers, and values may be written into arbitrary non-abstract stack slots. Thus, it is not necessary to replace a value on the stack via a sequence of pushes and pops. Rather, the element can be directly overwritten.

Additional mechanisms in the stack-typing discipline of TALx86 support other compiler tasks. For instance, to compile Popcorn exceptions, the code generator needs to pop off a dynamic amount of data from the control stack. To support this, TALx86 provides a limited form of pointers into the middle of the stack. These limited pointers are also sufficient to support displays (static links) for compiling languages such as Pascal. However, they are not sufficient to support general stack-allocation of data.

4.3 Memory Allocation

To support general heap allocation of data, TALx86 provides additional constructs which we now explore, beginning with tuples. Recall our Popcorn tuple code from Section 3:

```
*(int,int) x   = new (3, 4);
*(int,int) dbl = new (x.1+x.1, x.2+x.2);
```

At the assembly level, creating a new pair involves two separate tasks: allocating memory and initializing the fields. This TALx86 code corresponds to the preceding Popcorn:

```
malloc  8,<[:B4,:B4]> ; get space for x
mov     [eax+0],3     ; initialize x.1
mov     [eax+4],4     ; initialize x.2
push    eax           ; save x
malloc  8,<[:B4,:B4]> ; get space for dbl
mov     ebx,[esp+0]   ; x   in ebx
mov     ecx,[ebx+0]   ; x.1 in ecx
add     ecx, ecx      ; x.1+x.1 in ecx
mov     [eax+0], ecx  ; initialize dbl.1
mov     ecx,[ebx+4]   ; x.2 in ecx
add     ecx,ecx       ; x.2+x.2 in ecx
mov     [eax+4], ecx  ; initialize dbl.2
```

The `malloc` "instruction" is actually a macro that expands to code that allocates memory of the appropriate size. This routine puts a pointer to the newly-allocated space into `eax`. The verifier then knows that `eax` contains a pointer to *uninitialized* fields as specified in the typing annotation `<[:B4,:B4]>`.

Tracking initialization is important for safety because fields may themselves be pointers, and the type system should prevent dereferencing an uninitialized pointer. To do this, the type of every field in a piece

```
int sum(int n) {          sum: ∀ρ:Ts. {esp: sptr{eax: B4, esp: sptr B4::ρ}::B4::ρ}
  if (n==0)                   cmp     [esp+4],0
    return 0;                 jne     tapp(iffalse, <ρ>)
  else                        mov     eax,0
    return n+sum(n-1);        retn
}                         iffalse: ∀ρ:Ts. {esp: sptr{eax: B4, esp: sptr B4::ρ}::B4::ρ}
                              mov     ebx,[esp+4]
                              dec     ebx
                              push    ebx
                                ; recursive call instantiates ρ using current stack shape
                              call    tapp(sum, <{eax: B4, esp: sptr B4::ρ}::B4::ρ>)
                              add     esp,4
                              add     eax,[esp+4]
                              retn
```

Figure 1: Recursive Function with C Calling Convention

of memory has a variance, one of `u`, `r`, `w`, or `rw`, standing for uninitialized, read-only, write-only, and read-write respectively. The type system does not allow uninitialized fields to be read. However, uninitialized fields may be written with a value of the appropriate type and then the field is changed to a read-write field. Subtyping allows a read-write field to be used as read-only or write-only.

Here are the first three lines of our example where the comment describes the type that the verifier assigns to `eax` after each instruction:

```
malloc  8,<[:B4,:B4]> ; ^*[B4^u, B4^u]
mov     [eax+0],3     ; ^*[B4^rw, B4^u]
mov     [eax+4],4     ; ^*[B4^rw, B4^rw]
```

For example, the second type says, "a pointer to a tuple with two fields, an initialized `B4`, followed by an uninitialized `B4`". Of course, these pointer types can appear anywhere `B4` can, such as in part of a stack type or label type.

TALx86 places no restrictions on the order in which fields are initialized, nor does it require that all fields be initialized before passing the pointer to another function. It is possible for a field to be "initialized" more than once by creating an alias. For example:

```
malloc  8,<[:B4,:B4]>
mov     ecx, eax      ; ecx aliases eax
mov     [eax+0],3     ; init 1st field
mov     [ecx+0],4     ; init it again
```

In this code, when the contents of `eax` are moved into `ecx`, `ecx` is assigned the same type as `eax`. The two stores thus initialize the same field twice. However, this does not lead to a type unsoundness because the two values have the same type. Since the type system does not track aliasing, some semantically meaningful optimizations cannot be expressed in code that passes the verifier. For instance, the verifier rejects the following code because it assumes that field `[ecx+0]` is uninitialized:

```
malloc  8,<[:B4,:B4]>
mov     ecx, eax       ; ecx aliases eax
mov     [eax+0],3      ; init 1st field
mov     ebp,[ecx+0]    ; error!
```

Though it would be possible to augment TALx86 to conservatively track aliasing, doing so would further complicate the type system. Thus far, we have favored this simpler approach.

Finally, though TALx86 supports explicit allocation and deallocation of stack-allocated objects, it does not support general purpose pointers to stack-allocated objects. In contrast, general purpose pointers to heap-allocated objects are supported, but explicitly freeing them is not. Rather, we link the TALx86 code against a conservative garbage collector so that unreachable objects may be reclaimed. To support explicit freeing would require an extensive change to the type system [5].

4.4 Arrays

Support for arrays in TALx86 is perhaps the most complicated feature in the language. The critical issue is that array sizes and array indices cannot always be determined statically, yet to preserve type-safety, we must ensure that any index lies between 0 and the physical size of the array. Currently, TALx86 provides a very flexible mechanism for tracking the size of an array without requiring the size be placed in a pre-determined position (explained below).

Array subscripting and update require special macro instructions (`asub` and `aupd`) which take an array pointer, the size of the array, an integer offset, and for `aupd`, a value to place in the array. The macros expand into code sequences that perform a bounds check, exit immediately when the index is out of bounds, and otherwise perform the appropriate subscript or update operation. Because the array bounds checks are not separated from the subscript or update operations, an optimizer cannot eliminate them. Furthermore, no pointers into the middle of arrays are allowed by the current type system.

To support arrays, the TALx86 type system includes two new type constructors. The first, `S(`s`)`, is called a *singleton* type constructor, where s is a compile-time expression corresponding to an integer. The primary purpose of singleton types is to statically track the actual integer value of a register or word in memory. For instance, if `eax` has type `S(3)`, then the value in `eax` must be equal to 3 (i.e., it is drawn from the singleton set $\{3\}$). As with other kinds of type expressions, integer type expressions can be polymorphic. Thus, if `ecx` has type `S(`α`)`, then we cannot determine statically the (integer) value contained in `ecx`. However, if `ebx` also has type `S(`α`)`, then the type system can conclude that the contents of the two registers are equal. The type system treats singleton integer types as subtypes of `B4` so that they may be used whenever a `B4` is required.

The second new type constructor is of the form `array(`s`,`τ^v`)` where τ is the type of the array elements, v is their variance, and s is a type expression that represents the size of the array. Notice that s could be a constant, in which case the size of the array is known statically, or it could be a type variable, in which case the size of the array is unknown. Furthermore, as with other type expressions, s is a purely *static* construct used only for verification — it is *not* available as a runtime value. As we shall show, this gives us the flexibility to place the runtime array size anywhere we want instead of in some fixed position. Furthermore, if the size of the array can be determined statically, then the size need not be tracked at runtime.

The crucial issue is to enforce the property that *only a runtime integer value equal to the size of the array is passed to* `asub` *or* `aupd` *for the appropriate bounds check.* In particular, if the array has type `array(`s`,`τ^v`)`, then the integer value passed as the size of the array must have type `S(`s`)`. For example, the following TALx86 code increments index 2 of a size 5 array of `B4` values:

```
lab: {eax: array(5, B4^rw), ebx: S(5)}
    mov  ecx, 2
 ; put eax[ecx] into edx.
 ; array size in ebx, element size is 4.
    asub edx, eax, 4, ecx, ebx
    inc  edx
 ; put edx into eax[ecx].
 ; array size in ebx, element size is 4.
    aupd eax, 4, ecx, edx, ebx
```

This example may only be used on arrays of size 5. To support arrays whose size is unknown statically, we must introduce an integer type variable and quantify over it to achieve "size polymorphism":

```
lab:∀s:Sint.{eax: array(s,B4^rw), ebx: S(s)}
```

(The instructions do not need to change.)

Our compiler represents all Popcorn arrays as a pointer to a data structure containing the (runtime) size followed by the array elements. An *existential* type is used to tie the type of the runtime size with the type of the array as in:

```
∃s:Sint.^*[S(s)^r,array(s,B4^rw)]
```

The type reads as "there exists some integer s such that, I am a pointer to a struct containing an integer equal to s, followed by s `B4` values." Using an existential to package the runtime size with the array, we can pass the data structure to any function, or place it in any data structure and yet maintain enough information that we can always perform a checked subscript or update on the array. Notice that though this is the default representation used by the compiler, it is not required by TALx86. In particular, the runtime size and the underlying array may be "unboxed" when the Popcorn array does not escape. In situations where the size of the array is known at compile time, an optimizer could avoid storing the size entirely.

Finally, there are two ways to create arrays in TALx86. An n-tuple of values, all of some type τ and variance v, may be coerced to an array of type `array(`n`,`τ^v`)`. Second, the trusted runtime provides a function which takes an integer n and a value x of type τ and returns an array of size n with each array element initialized to x.

Currently, we are working to eliminate the `asub` and `aupd` macros and to expose the bounds checks so that an optimizer may eliminate them. To do so requires supporting a more expressive symbolic language of static integer expressions within the type

```
?struct int_list {
  int hd;
  int_list tl;
}
int len(int_list lst){
  int i = 0;
  while (lst != null){
    ++i;
    lst = lst.tl;
  }
  return i;
}
```

```
type   <int_list:T4 = ^.(0) *[B4^rw,`int_list^rw]>

len: ∀ρ:Ts.
   {esp: sptr{eax: B4, esp: sptr `int_list::ρ}::`int_list::ρ}
   mov    eax, 0                       ; i=0  in eax
   mov    ebx, [esp+4]                 ; lst in ebx
   jmp    tapp(test, <ρ>)
body: ∀ρ:Ts.{esp: ..., eax: B4, ebx: ^*[B4^rw,`int_list^rw]}
   inc    eax                          ; ++i
   mov    ebx, [ebx+4]                 ; lst = lst.tl
   fallthru <ρ>
test: ∀ρ:Ts.{esp: ..., eax: B4, ebx:`int_list}
   coerce unroll(ebx)   ; int_list -> ^.(0) *[B4^rw,`intlist^rw]
   btagi  ne, ebx, 0, tapp(body,<ρ>) ; check if ebx is null (0)
   retn                                ; otherwise return
```

Figure 2: List of Integers Implementation

system and the ability to prove inequalities between such expressions as with Xi and Pfenning [19, 20].

4.5 Sums and Recursive Types

To demonstrate TALx86 sums and recursive types, we now consider implementing a linked list of integers (see Figure 2). There are two critical points here: First, a list is fundamentally a *sum type*: a value of type list is either null or a pointer to a tuple, and we must ensure that the code works in either case. Second, the type of list is recursive.

The Popcorn code has a `?struct` definition for lists and a `len` function which, when given a list, calculates its length. The TALx86 code has a corresponding type definition and corresponding code. The TALx86 type definition says a value can be coerced to have type `int_list` if it is either the singleton value 0 (for `null`) or a pointer to a pair of an integer and an `int_list`.

Upon entry to the `len` label, the integer variable i is initialized to 0 and placed in register `eax`. The list argument is placed in `ebx` and the code jumps to the loop test. The test coerces `ebx` from the type `int_list` to its representation type, namely the corresponding sum type. The next instruction, `btagi`, is a macro instruction that tests whether `ebx` is not equal (`ne`) to 0, and if so, branches to the body. The macro expands into a simple compare and branch. The type-checker verifies that the register being tested has a sum type, and using the value tested against, refines the type of the register. In particular, at the label `body`, we are allowed to make the stronger assumption that `ebx` is in fact a pointer, and not null. This assumption allows the `mov ebx,[ebx+4]` operation to verify, which has the effect of setting `ebx` to the tail of the list.

Our current Popcorn compiler generates more naïve code: The list is tested for null once as part of the while test, and then again when the tail of the list is selected. However, it is clear that an optimizing compiler can do dataflow analysis to determine that the second check is redundant. What is not as clear is whether an optimizing compiler can easily maintain the appropriate typing annotations.

4.6 Making Types Smaller

The TALx86 type annotations take far less space than we have suggested so far. For example, the verifier allows the typing preconditions to be dropped for certain labels. In particular, labels that serve only as forward branch targets need no typing precondition. The verifier simply re-typechecks the corresponding code block for each branch. The restriction to forward branches ensures termination of the verifier.

The verifier also supports type abbreviations so that the common sub-terms of types may be abstracted. For example, Popcorn gives the same type to every `string`. Rather than repeat this type everywhere, Popcorn defines a `str` abbreviation and uses it in place of the unabbreviated form:

```
type <str = ∃s:Sint.^*[S(s)^r,array(s,B1^rw)]>
```

Another source of repetition is the code types. For example, our code types essentially repeat the type of the stack twice, once for the stack and once for the type of the return address. We can abstract the calling convention with a function abbreviation:

```
type <F = fn ret:T4 s:Ts.
     {esp: sptr {eax: ret, esp: sptr s}::s}>
```

For example, the fully expanded type of the polymorphic map function is the rather unwieldy:

```
map: ∀α:T4 β:T4 ρ:Ts.
{esp: sptr
  {eax:(‘list β),
   esp:sptr(∀ρ′:Ts.
            {esp: sptr{eax:β esp: sptr α::ρ′}
                        ::α::ρ′})
            ::(‘list α)::ρ}
  ::(∀ρ′:Ts.
    {esp: sptr{eax:β esp: sptr α::ρ′}
              ::α::ρ′})
  ::(‘list α)::ρ}
```

but with the above abbreviation becomes:

```
map: ∀α:T4 β:T4 ρ:Ts.
     F (‘list β)
       ((∀ρ′:Ts. F β (α::ρ′))::(‘list α)::ρ)
```

which is smaller, more readable, and in practice faster to verify.

5 Summary and Future Work

We have described the currently available tools for producing TALx86, including a compiler for the C-like language Popcorn. Through examples, we have demonstrated how TALx86 can ensure the safety of assembly code, even in the presence of advanced structures and optimizations.

Planned extensions to our system will both add tools and increase the expressiveness of the languages. They include:

1. A binary object file format to replace TALx86's current ASCII format. This will save both space and parsing time.

2. Support for floating point and MMX instructions. We do not expect this to be difficult.

3. Support for run-time code generation, as developed by Trevor Jim and Like Hornoff at the University of Pennsylvania [7]. In addition, an extension to Popcorn called Cyclone makes these features available at a higher level. We are currently working through some minor interoperability issues.

4. Object support in the form of subtyping, bounded quantification, and self-quantification. These features are required to compile object-oriented languages effectively and safely.

5. A more advanced dependent type system to eliminate bounds checks when it can be proven that it is safe to do so.

6 Acknowledgments

We thank Chris Hawblitzel who wrote the first version of Popcorn. Trevor Jim and Luke Hornoff provided invaluable feedback on the system in addition to adding support for runtime code generation.

References

[1] Martín Abadi and Raymie Stata. A type system for Java bytecode subroutines. In *Twenty-Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 149–160, San Diego, California, USA, January 1998.

[2] Nick Benton, Andrew Kennedy, and George Rusell. *The MLJ User Guide*, 1998.

[3] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[4] Karl Crary. A simple proof technique for certain parametricity results. Technical Report CMU-CS-98-185, Carnegie Mellon University, December 1998.

[5] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Twenty-Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, Texas, USA, January 1999.

[6] Neal Glew and Greg Morrisett. Type safe linking and modular assembly language. In *Twenty-Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 250–261, San Antonio, Texas, USA, January 1999.

[7] Luke Hornof and Trevor Jim. Certifying compilation and run-time code generation. In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 60–74, San Antonio, Texas, USA, January 1999.

[8] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, September 1998. With H. Abelson, N. I. Adams,

IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, and M. Wand.

[9] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1988.

[10] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[11] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Workshop on Types in Compilation*, pages 95–118, Kyoto, Japan, March 1998. Published as LNCS 1473, pages 28–52, Springer-Verlag.

[12] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language (extended version). Technical Report TR97-1651, Department of Computer Science, Cornell University, 4130 Upson Hall, Ithaca, NY 14853-7501, USA, November 1997.

[13] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Twenty-Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego California, USA, January 1998.

[14] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Twenty-Fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 132–145, Paris, France, 1997.

[15] George Necula. Proof-carrying code. In *Twenty-Fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 1997.

[16] Robert O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Twenty-Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 70–78, San Antonio, Texas, January 1999.

[17] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Twenty-Fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, January 1997.

[18] OOPSLA'98 Workshop. *Formal Underpinnings of Java*. Vancouver, Canada, October 1998.

[19] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, Canada, June 1998.

[20] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Twenty-Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, Texas, USA, January 1999.