Typed Assembly Language

Brett Giles

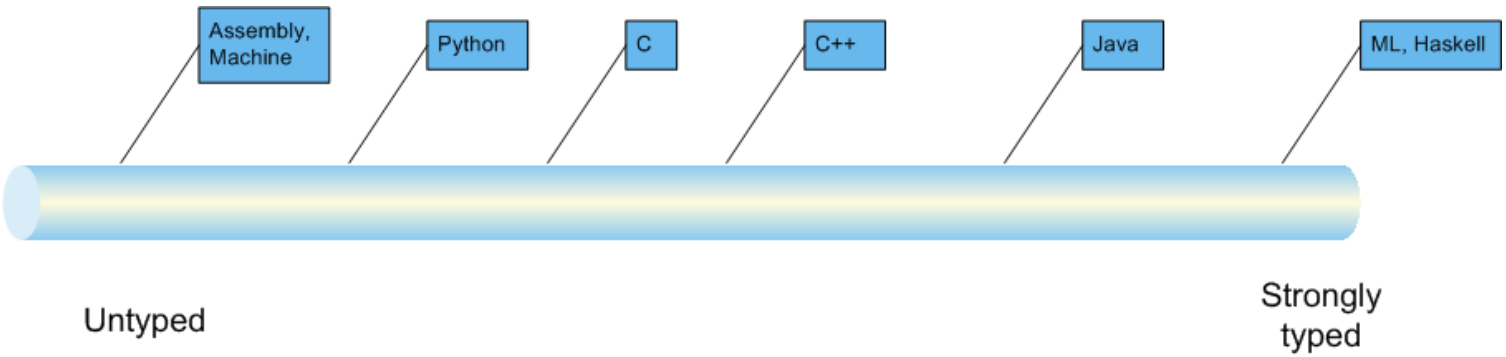
Department of Computer Science
University of Calgary

2008-01-17 / Term presentation

Outline

What is a Typed Language?

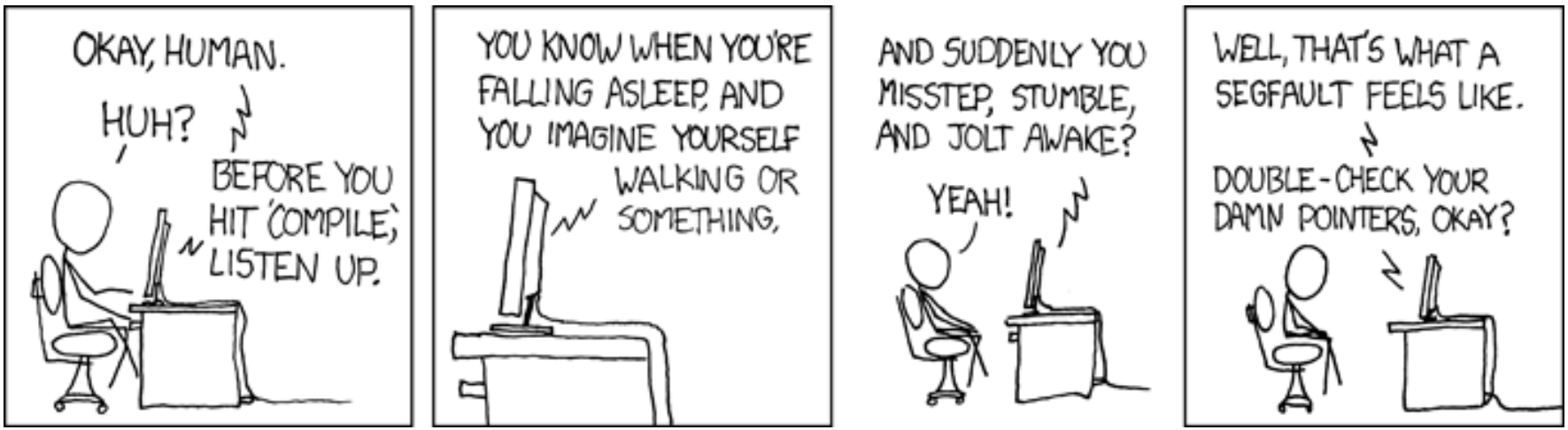
Languages can be placed on a continuum related to their support for typing.



Typically, in a typed language, you can predict the types of all values at an execution point.

Why do we care?

Think of your poor computer. . .



What is a Typed Assembly Language?

- A low-level language that adds type information, moving it to the right on the typing continuum. Example characteristics:
  - An integer is recognizably different from a pointer.
  - A character is different from a Boolean.
- The Java Virtual Machine and its bytecode.
- .NET bytecode.

Why TALs are interesting

Executing untrusted code can be a risky experience.

This happens more and more with the Internet. Typed Assembly Languages allow safety properties to be *proved* about them.

This was one of the historical drivers for Java applets.

History of TAL Research

Arguably, the JVM was the first TAL, and was specified in the mid 1990's.

This was followed by a series of papers in the late 90's by people at Cornell and CMU investigating the theory of low level languages and types.

Following that, the same people created a TAL language for the x86 architecture and a type-safe C-like language.

More recent research is starting to specialize TALs, e.g., for garbage collection, translations to certified assembly programming and other ways to ensure safety of low-level programs.

Relationship to compiler technology

Typically, at some point, compilers begin to erase (lose) type information. With TAL:

- Types are preservable to a very low level.
- Optimizations such as CSE, loop unrolling, etc. are still available.
- Some optimizations are not, e.g., elimination of array bounds checking.

Related research

As mentioned in the history slide, a stated goal of this research is proving safety properties of programs. This goal is shared by:

- Proof Carrying Code;
- Certified Assembly Programming;

and supported by:

- Type theories for TALs, semantics of specific TALs;
- Certifying compilers.

TAL

The original version, introduced by Morrisett et. al. as a type preserving series of translations from System F (typed Lambda Calculus with values) to TAL.

TAL is a generic RISC-type assembly language, with the addition of type annotations. All code labels are annotated with the required/expected types at entry to the label. All heap data is annotated with its type, which includes information about initialization.

Data is allocated only in the heap by *malloc* which is considered an atomic operation that allocates space and sets the type.

STAL - Stack based TAL

Stack based TAL is an extension of TAL that provides typing for stack operations. This allows stack based local variables and activation records.

In contrast to the JVML, procedures, exceptions and calling conventions are not part of the specification of STAL. This allows a language designer to choose those best suited to the language.

The main addition to the type system is a form of stack polymorphism supporting polymorphic recursion.

TALx86 - TAL for the IA-32

With a type system based upon that of STAL, TALx86 is an assembly language for the x86 brand of CPUs. The practical typing aspects of the language are:

- Interface information to allow separate type-checking.
- Type constructor declarations
- Type preconditions on code labels
- Types on data labels
- Type coercions on operands
- Macro instructions, e.g., *malloc*

TALT - TAL Two

Introduced by Crary in 2003, this TAL enhances TALx86.

The type system for TALT is substantially advanced over its predecessors, supporting constructs such as heterogeneous tuples, arrays and recursive types via the type theory, rather than by annotations and special code in a type checker.

The goals for this TAL include:

- Sufficient expressive to be the target of certifying compilers;
- Operational semantics that map well to machine operations;
- Machine checking of the type safety of a program.

GTAL - Garbage Collecting TAL

In this paper, Chen et. al. enhance the type system of a TAL to prove the safety of a garbage collector written in the language.

The main reason for doing this is to reduce the base of programs that must be *trusted* when proving the safety of programs. Typically, implementations of TALs rely on a run-time system that has a trusted base of code. In this context trust means you believe it works even though you were not given a mechanical proof of this fact.

TAL typing

Type syntax:

$$
\begin{array}{rcl}
\text{types} & \tau & ::= \alpha \mid \mathit{int} \mid \langle \tau_1, \ldots, \tau_n \rangle \mid \forall[\Delta].\Gamma \mid \exists \alpha.\tau \\
\text{type assignments} & \Delta & ::= \langle \mathit{empty} \rangle \mid \Delta, \alpha \\
\text{registers} & \Gamma & ::= \{r_1 : \tau_1, \ldots, r_n : \tau_1\} \\
\text{labels} & \Psi & ::= \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_1\}
\end{array}
$$

With the main typing judgement being:

$$\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi; \cdot; \Gamma \vdash I}{\vdash (H, R, I)}$$

The rest of the syntax defines registers, values, heaps and instructions.

Adding a Stack to TAL

- Model a stack as a list of word-sized values.
- Add instructions to allocate / deallocate stack space.
- Add instructions to load from / store to the stack.
- Add a distinguished register *sp* to point to the stack.

Type additions are:

$$\begin{array}{rrl} \text{stack types} & \sigma & ::= \rho \mid \mathit{nil} \mid \tau :: \sigma \\ \text{stacks} & S & ::= \mathit{nil} \mid w :: S \end{array}$$

with minor adjustments to the rest of the syntax.

Stacks and Polymorphism

As mentioned when discussing STAL, many compilers generate stack based code, specifically for local variables and activation records.

A requirement of typing is that the stack be properly typed upon entry to any routine. This requires polymorphism. A typical typing will be:

$$\forall[\rho].\{r1 : int, sp : int :: \rho, rrtn : \{r1 : int, sp : \rho\}\}$$

This forces the stack to have an integer parameter at entry, which the routine must remove. The rest of the stack must be the same on exit as on entry. This allows the stack to contain anything else at entry and allows the creation of local storage.

Polymorphism for branching

Consider:

```
cd∀[α] {r1:α,r2:{r1:⟨α,α⟩}}.
       malloc   r3,⟨r1,r1⟩
       mov      r1,r3
       jmp      r2
```

This allows jumping to 'cd' with any value of any type in $r1$. It will then be duplicated, the pointer to the tuple moved into $r1$ and we return.

Space allocation and initialization

As we saw in the previous example, TAL includes an instruction (macro) named `malloc`. In STAL, this operation is treated as an atomic allocation of space together with initialization of the values as supplied to it.

It is possible to split the allocation and initialization steps, as was done in TAL, but at the expense of complicating the type system to track uninitialized values.

TALT is closer to the original TAL, with the `malloc` being part of the run-time trusted base, allocating *n* bytes and putting the resulting pointer to the junk values in a destination.

Arrays

In TALx86, two instruction macros, `asub` and `aupd` are added to get/update array elements. Additionally the type system adds the concept of singleton types (e.g, the type $S(3)$ is inhabited only by the number 3), and a type constructor for arrays.

This is sufficient to support fixed and dynamic sized arrays and support atomic bounds checking. Note that bounds checking is unoptimizable in this scenario.

Arrays in TALT

In TALT, arrays are handled substantially in the same way, except that the array access and update instructions do not include bounds checking. Instead, the type system requires the bounds to be checked before access. This means the compiler may eliminate bounds checks where they are not needed, for example, a constant offset into a fixed size array.

Popcorn - a safe C

Think C / Java + types with type safety.

The TAL people designed this language so that it would attract the C / Java programmer, yet force true typing and type safety on them. The language adds type constructors and treats `struct` and `union` as type declarations.

A compiler for Popcorn to TALx86 is written in ocaml and in Popcorn. Unfortunately, it currently doesn't compile on Linux. No new versions have been posted on the site since 2003. At that point, the declared intention was to move to cyclone.

Cyclone

A language based upon the C9X OSI standard, with additions to handle typing while deviating as little as possible from the standard.

Unfortunately, the compiler for Cyclone is written in Popcorn and I haven't been able to get it to compile.

Other languages

Initially, some work was done on a scheme subset compiler. The documentation says that no longer works.

In principle, any strongly typed language (ML, Haskell, ocaml, scheme,...) could be compiled to TAL. Languages such as C, Python, and Java could be problematic as the compiler would, in some cases, have to actually add type information not given in the program. Type inference might be able to handle that in some cases.

A recursive program

```
sum: ∀ρ:Ts. {esp: sptr{eax: B4, esp: sptr B4::ρ}::B4::ρ}
    cmp     [esp+4],0
    jne     tapp(iffalse, <ρ>)
    mov     eax,0
    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 stk shape
    call    tapp(sum, <{eax: B4, esp: sptr B4::ρ}::B4::ρ>)
    add     esp,4
    add     eax,[esp+4]
    retn
```

Observations

The initial results of the TAL papers (TAL, TALx86) seem somewhat unfinished, with a variety of vexing issues, ranging from macros that atomicize important issues to requiring actual annotation of the code. While they were interesting papers, there seemed to be something missing.

I believe TALT changes this with a much stronger type system and generally better organization. Karl Crary, the authour of TALT, is a part of the ConCert project investigating Certified Code for Grid Computing, where he is exploring the use of TALT as part of the project.

Summary

- TALs are an interesting combination of type theory and compiler writing.
- TALs may be of great use in certifying code for running over the Internet from untrusted sources.