Summary of "Typed Assembly Language"[1]

CHRISTOPH MENDE, NIKOLAJ-JENS SCHWARTZ

1 INTRODUCTION

We want to have a machine-checkable proof, that code has some desired properties. For that we want to use proof carrying code. There are two problems to solve: Firstly, what properties should we require of the code? Secondly, how do code producers construct a formal proof, that their code has the desired properties? The first is extremly context and application dependent. The second is impossible to solve automatically for arbitrary code. One approach is type-preserving compilation. High level programming languages check for type safety. With the help of a type-preserving compiler the type-safe source code is broken down through lower-level intermediate languages to target code. In this process the proof is transformed as well. As there are pre-conceived notions and properties of methods and objects, that may be incomaptible for a given language. So the chance of having a typed intermediate language (TIL) capable of handling all sorts of high level languages and implementation strategies is improbable. The way to go is to design a TIL that minimizes the need to add new features and typing rules.

2 TAL-0: CONTROL-FLOW-SAFETY

As we do not want a programm to jump to random adresses when it's executed, control-flow safety is crucial when using dynamic checks in a system. We start with a simple abstract machine. Each instruction uses the value in a source register r_s an operand n (integer literal), l (label or pointer) or r (register) to compute a value, which is placed in the destination register r_d. A *value* is an operand that is not in the register. I is defined as a list of instructions. Instead of using a concrete machine, we take an abstract machine which has certain distinctions such as the distinction between labels and arbitrary integers. This helps to state and prove that control-flow instructions can only jump to valid entry points. If we try to transfer control to an integer instead of a label, the abstract machine will get stuck. Hence our main goal is to prevent our machine from getting stuck.

3 THE TAL-0 TYPE SYSTEM

The TAL-0 Type System tries to ensure that any state M always has a following state M' and as such can never get stuck. To ensure that, the type system needs to distinguish between labels and integers to make sure that the control flow follows labels and that the typing is preserved so that we cannot reach a stuck state after transferring control to a label.

The type system in our TAL classifies four different type constructors. These are int, $code(\Gamma)$, α and $\forall\alpha.\tau$. The first type is int for simple integer values. The second, $code(\Gamma)$ is for labels and the types that are expected in the registers. Γ is a *register file type*, a total function that maps every register to a specific type. The last two types are for polymorphism, α is a type variable that can be substituted for any type and $\forall\alpha.\tau$ is a term that includes the type variable α.

Additionally, our type system includes a Ψ that maps every label to a type.

With the typing rules defined in figure 4-4, it is now possible to show that the register file type Γ is never changed.

It is also shown that polymorphism is needed as soon as we assign a label to a register and jump to that label, because the type of the register — for example register $r1$ — would be $code(\Gamma)$, but that $code(\Gamma)$ would also include $r1$. The result would be a recursive type. To avoid this, the type of $r1$ inside $code(\Gamma)$ is defined as α.

Another use of polymorphism is to enforce a function to not change the contents of a register. If Γ is defined in a way that expects $r1$ to be of type α when a label is entered and also after returning, it is not possible for the code under that label to change the contents of $r1$. Any assignment to $r1$ would change the type from α to some concrete type that would not match the expected α on return.

3.1 Proof of Type Soundness for TAL-0

The proof of soundness includes two proofs. To be sound, TAL-0 must be able to progress — that is, any well-formed state M must have a successor state M' — and it must preserve well-formedness — that is, the successor state M' is also well-formed if the initial state M was well-formed.

To do this, a few lemmas are defined and the rules in figure 4-4 are used. The actual proof then proceeds by induction on the instruction sequences I. I is a good candidate for induction, because it is defined in a way similar to the natural numbers with 0 being $jump\ v$ and the successors of 0 being instructions prepended to $jump\ v$.

3.2 Proof Representation and Checking

4 TAL-1: SIMPLE MEMORY-SAFETY

5 EXERCISE 4.2.1

Proof that $r3 := 0$ preserves Γ:

$$\dfrac{\dfrac{}{\Psi;\Gamma \vdash r3 : \Gamma(r3) = int}\ \text{S-REG} \qquad \dfrac{}{\Psi \vdash 0 : int}\ \text{S-INT}}{\Psi \vdash r3 := 0 : \Gamma \to \Gamma}$$

Proof that $jump\ loop$ preserves Γ:

$$\dfrac{\dfrac{\dfrac{}{\Psi;\Gamma \vdash loop : \Psi(loop) = code(\Gamma)}\ \text{S-LAB}}{\Psi;\Gamma \vdash loop : code(\Gamma)}\ \text{S-VAL}}{\Psi \vdash jump\ loop : code(\Gamma)}$$

Proof that $jump\ r4$ preserves Γ:

$$\dfrac{\dfrac{\dfrac{}{\Psi;\Gamma \vdash r4 : \Gamma(r4) = code(\Gamma)}\ \text{S-LAB}}{\Psi;\Gamma \vdash r4 : code(\Gamma)}\ \text{S-REG}}{\Psi \vdash jump\ r4 : code(\Gamma)}$$

REFERENCES

[1] Benjamin C. Pierce. 2004. *Advanced Topics in Types and Programming Languages.* The MIT Press.