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 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:

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

5 TAL-1: SIMPLE MEMORY-SAFETY

TAL-1 extends TAL-0 with the support to share objects by reference. The language needs memory safety, to assure that the programm can only access objects, that it has been granted access to. In addition we need a location where we can store values of different types at different times. At least we need the possibility to support the construction of compound values. While on assembly level the allocation can be done in one expression, we need several steps on machine level. At first we need to allocate space for the object, then initialize the components by storing them in the allocated space. Depending on the status of initialization, we use different types with the purpose to only access fully initialized valid values. If there are data objects in the registers, changes can not be tracked. This is also true for every register, which holds an alias fron a changed value, e.g. pointers. For this approach the type system needs to check if two values are the same, or two labels act like they are the same. Furthermore we need support for allocating and initializing data structures that are to be shared and stack-allocating procedure frames. For implementing type safe, high level languages on conventional architectures we need the type system to be separated into two classes. One for *shared pointers*, which allows

arbitrary aliasing, but with the constriction, that the contents must remain invariant. The second class is for *unique pointers*, which supports changing the contents type. Restricted in the way, that these pointers are not allowed to have aliases und must not be copied. The combination of shared and unique pointers provides us with a simple but flexible framework for memory management. With this at hand, we are able to handle problems of allocating and initializing shared data structures. We can also handle compiler controlled lifetime of data structures like stack frames with the help of unique pointers.

6 TAL-1: EXTENDED ABSTRACT MACHINE

TAL-0 is extended with a set of six instructions. Two load values from memory to registers, or the other way around. The instruction used to load from the memory behaves different, whether the source register holds a shared or a unique value. I case of a shared value we have to look up the value in the heap space, while the unique value is available instant. *malloc* instruction is used to allocate n Words. A unique reference to the object is placed in the destination register. *commit* is used to convert a unique pointer into a shared pointer. This has no real effect, but helps us to state and prove the invariants of type system. The *salloc* and *sfree* constructs manipulate a special unique pointer, which is held in a distinguished register, called stack pointer. *salloc* adds n words to the stack and *sfree* deletes n words from the stack. The type system prevents the stack from underflowing, but not from overflowing. Thus we expect the *salloc* instruction to check for overflow an abort the computation. The machine states are modeled via a triple of a heap, register file and instruction sequence. - Register files map registers to word-sized values - heaps map labels to heap-values Heap-values are extended to include tuples of word-sized values. Therefore a label can refer to code or data. Operands are extended with terms, which represent a unique pointer to a heap value. The rewriting rules for instructions of TAL-1 are mostly the same with TAL-0. But we have to prevent that unique pointers are copied or references point to the same data.

7 THEOREM 4.2.10 FOR TAL-1

case $I = r_d := v; I'$: From inversion of the S-MOV-1 rule, we have $\Psi \vdash r_d := v : \Gamma \rightarrow \Gamma[r_d : \tau]$, $\Psi; \Gamma \vdash v : \tau$ and $\tau \neq uptr(\sigma)$ for some τ. By the Register Substitution lemma, we have $\Psi; \Gamma \vdash \hat{R}(v) : \tau$. Taking $M' = (H, R[r_d = \hat{R}(v)], I')$, we see that $M \rightarrow M'$ via the MOV-1 rule. From the S-REGFILE rule, we conclude that $\Psi \vdash R[r_d = \hat{R}(v)] : \Gamma[r_d : \tau]$.

case $I = malloc\ n; I'$: From inversion of the S-MALLOC rule, we have $\Psi \vdash r_d := malloc\ n : \Gamma \rightarrow \Gamma[r_d : uptr\langle \underbrace{int, ..., int}_{n} \rangle]$ and $n \geq 0$. Taking $M' = (H, R[r_d = uptr\langle m_1, ..., m_n \rangle], I)$, we see that $M \rightarrow M'$ via the MALLOC rule.

REFERENCES

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