Abstract. Machine Deduction is a type system designed to extract code for an abstract machine from proofs. This paper presents the basic definitions and results, and shows how we can replace a compilation of typed lambda-terms by a proof translation into our system.

1 Introduction.

The proofs as programs paradigm, using the Curry-Howard isomorphism [4], gives a way to associate a program to an intuitionistic proof. This program is almost always a functional program (in general a lambda-term [1]) which has to be compiled before being executed [15]. This ensures some correctness about the functional program extracted from the proof. But the correctness of the compiled code is relative to the proof of the compiler.

We study in this paper a type system for the code of an abstract machine (S.E.C. machine or Krivine's machine). This approach authorises a new kind of compilation: we translate the proof in natural deduction to a proof in our system and we extract the code from this new proof.

The two kinds of compilation can be represented by the following diagram:

To achieve this goal we define a deduction system $MD_{SEC}$, for intuitionistic logic. This system is a second order system specially tailored to a translation of Leivant and Krivine's system AF$_2$[5, 8, 12, 9].

Secondly, we define an S.E.C. machine which is a modification of Krivine's machine (using arrays to store environment and adding some instructions to save

* email: cr@dcs.ed.ac.uk
and restore the stack). Then we introduce a semantics for intuitionistic logic in terms of this machine and we show how to extract code from a proof in our system in a way which is sound for this semantics.

Next, we show how to deal with data types, and how the previous notion of semantics ensures the correctness of programs on data types. In fact $MDSEC$, may be seen as an extension of AF2 in which terms are replaced by compiled code for the S.E.C. machine. However, the ability to use machine data types is the principal benefit of $MDSEC$, because it allows us to consider the notion of computed data inside the system, with an associated type (see section 8).

Finally we give two proof translations, one for Krivine's call-by-name compilation of classical logic and another which is an alternative to call-by-value. This illustrates how our method ensures the correctness of the compiler through the use of data types like the natural numbers. We also briefly show how we can optimise our last compilation in this framework.

2 The deduction system.

We use classical second order formulas. First we define first order terms from a language $\mathcal{L}$ defined by an algebraic signature $\Sigma$. Then we define by induction two sets of formulas: $\mathcal{F}^\phi$, the set of value formulas and $\mathcal{F}^\Pi$ the set of stack formulas:

**Definition 1.** We choose some infinite sets of predicate variables: $\forall_n^\phi$ the set value variables of arity $n$ and $\forall_n^\Pi$ the set of stack variable of arity $n$. And we define $\mathcal{F}^\phi$ and $\mathcal{F}^\Pi$ as the least sets of word satisfying:

- $\top, \bot \in \mathcal{F}^\Pi$
- $X(t_1, \ldots, t_n) \in \mathcal{F}^\Pi$ if $X \in \forall_n^\Pi$
- $F \land P \in \mathcal{F}^\Pi$ if $F \in \mathcal{F}^\phi$ and $P \in \mathcal{F}^\Pi$
- $\exists X P \in \mathcal{F}^\Pi$ if $P \in \mathcal{F}^\Pi$ and $X$ is any kind of variable
- $A(t_1, \ldots, t_n) \in \mathcal{F}^\phi$ if $A \in \forall_n^\phi$
- $\neg P \in \mathcal{F}^\phi$ if $P \in \mathcal{F}^\Pi$

Here is some intuition about the meaning of these formulas: Formulas in $\mathcal{F}^\phi$ will type values and formulas in $\mathcal{F}^\Pi$ will type stacks of values. Thus an object of type $F \land P$ should be understood as a stack starting with a value of type $F$ and continuing with a stack of type $P$. A value of type $\neg P$ will be a program which terminates when it is applied to any stack of type $P$.

In all this paper, we will use the following notation to write formulas:

- $M, N$ for arbitrary formulas.
- $F, G$ for value formulas.
- $A, B$ for value variables.
- $t, u$ for first order terms.
- $M[t/x]$ for the substitution of the first order variable $x$ with a term $t$ in the formula $M$. 
- $P, Q, R$ for stack formulas.
- $X, Y$ for stack variables.
- $x, y$ for first order variables.
- $\Gamma, \Delta$ for multisets of value formulas.