Declarative Language Definitions and Code Generation as Linearization

Aarne Ranta
Department of Computing Science
Chalmers University of Technology and the University of Gothenburg
aarne@cs.chalmers.se

30 November 2004

Abstract

This paper presents a compiler for a fragment of the C programming language, with JVM (Java Virtual Machine) as target language. The compiler is implemented in a purely declarative way: its definition consists of an abstract syntax of program structures and two concrete syntaxes matching the abstract syntax: one for C and one for JVM. From these grammar components, the compiler is derived by using the GF (Grammatical Framework) grammat tool: the front end consists of parsing and semantic checking in accordance to the C grammar, and the back end consists of linearization in accordance to the JVM grammar. The tool provides other functionalities as well, such as decompilation and interactive editing.

Notice. *Revised 2 March 2010 for GF 3.1. The only change is that the GF parser is now used directly, instead of going via BNFC (Section 7).*

1 Introduction

The experiment reported in this paper was prompted by a challenge posted by Lennart Augustsson to the participants of the workshop on Dependent Types in Programming held at Dagstuhl in September 2004. The challenge was to use dependent types to write a compiler from C to bytecode. This paper does not meet the challenge quite literally, since our compiler is for a different subset of C than Augustsson's specification, and since the bytecode that we generate is JVM instead of his format. But it definitely makes use of dependent types.

Augustsson's challenge did not specify *how* dependent types are to be used, and the first of the two points we make in this paper (and its title) reflects our interpretation: we use dependent types, in combination with higher-order abstract syntax (HOAS), to define the grammar of the source language (here, the fragment of C). The grammar constitutes the single, declarative source from which the compiler front end is derived, comprising both parser and type checker.

The second point, code generation by linearization, means that the back end is likewise implemented by a grammar of the target language (in this case, a fragment of JVM).

This grammar is the declarative source from which the compiler back end is derived. In addition, some postprocessing is needed to make the code conform to Jasmin assembler requirements.

The complete code of the compiler is 300 lines: 250 lines for the grammars, 50 lines for the postprocessor. The code is presented in the appendices of this paper.

2 The Grammatical Framework

The tool we have used for implementing the compiler is GF, the *Grammatical Framework* [15]. GF is similar to a Logical Framework (LF) [4] extended with a notation for defining concrete syntax. GF was originally designed to help building multilingual translation systems for natural languages and also between formal and natural languages. The translation model implemented by GF is very simple:

```
             parsing                          linearization
             ------------>                    ------------>
  Language_1                Abstract Syntax                 Language_2
             <------------                    <------------
             linearization                    parsing
```

An abstract syntax is similar to a *theory*, or a *signature* in a logical framework. A concrete syntax defines, in a declarative way, a translation of abstract syntax trees (well-formed terms) into concrete language structures, and from this definition, one can derive both linearization and parsing.

To give an example, a (somewhat simplified) translator for addition expressions consists of the abstract syntax rule

```
  fun EAdd : (A : Typ) -> Exp A -> Exp A -> Exp A ;
```

the C concrete syntax rule

```
  lin EAdd _ x y = {s = x.s ++ "+" ++ y.s ; prec = 2} ;
```

and the JVM concrete syntax rule

```
  lin EAdd t x y = {s = x.s ++ y.s ++ t.s ++ "_add"} ;
```

The abstract syntax rule uses a type argument to capture the fact that addition is polymorphic (which is a simplification, because we will restrict the rule to numeric types only) and that both operands have the same type as the value. The C rule shows that the type information is suppressed, and that the expression has precedence level 2 (which is a simplification, since we will also treat associativity). The JVM rule shows how addition is translated to stack machine instructions, where the type of the postfixed addition instruction has to be made explicit. Our compiler, like any GF translation system, will consist of rules like these.

The number of languages related to one abstract syntax in a translation system is of course not limited to two. Sometimes just one language is involved; GF then works much the same way as any grammar formalism or parser generator. The largest number of languages in an application known to us is 88; its domain are numeral expressions from 1 to 999,999 [16].

In addition to linearization and parsing, GF supports grammar-based *multilingual authoring* [6]: interactive editing of abstract syntax trees with immediate feedback as linearized texts, and the possibility to textual through the parsers.

From the GF point of view, the goal of the compiler experiment is to investigate if GF is capable of implementing compilers using the ideas of single-source language definition and code generation as linearization. The working hypothesis was that it *is* capable but inconvenient, and that, working out a complete example, we would find out what should be done to extend GF into a compiler construction tool.

2.1 Advantages and disadvantages

Due to the way in which it is built, our compiler has a number of unusual, yet attractive features:

The front end is defined by a grammar of C as its single source.

The grammar defines both abstract and concrete syntax, and also semantic well-formedness (types, variable scopes).

The back end is implemented by means of a grammar of JVM providing another concrete syntax to the abstract syntax of C.

As a result of the way JVM is defined, only semantically well formed JVM programs are generated.

The JVM grammar can also be used as a decompiler, which translates JVM code back into C code.

The language has an interactive editor that also supports incremental compilation.

The problems that we encountered and their causes will be explained in the relevant sections of this report. To summarize,

The scoping conditions resulting from higher-order abstract syntax are slightly different from the standard ones of C.

Our JVM syntax is slightly different from the specification, and hence needs some postprocessing.

Using higher-order abstract syntax to encode all bindings is sometimes cumbersome.

The first shortcoming seems to be inevitable with the technique we use: just like lambda calculus, our C semantics allows overshadowing of earlier bindings by later ones. The second problem is systematically solved by using an intermediate JVM format, where symbolic variable addresses are used instead of numeric stack addresses. The last shortcoming is partly inherent in the problem of binding: to spell out, in any formal notation, what happens in complex binding structures *is* complicated. But it also suggests ways in which GF could be tuned to give better support to compiler construction, which, after all, is not an intended use of GF as it is now.

3 The abstract syntax

An *abstract syntax* in GF consists of `cat` judgements

$$\texttt{cat}\ C\ \Gamma$$

declaring basic types (depending on a context Γ), and `fun` judgements

$$\texttt{fun}\ f\ \texttt{:}\ A$$

declaring functions f of any type A, which can be a basic type or a function type. *Syntax trees* are well-formed terms of basic types, in η-long normal form.

As for notation, each judgement form is recognized by its keyword (`cat`, `fun`, etc), and the same keyword governs all judgements until the next keyword is encountered.

The abstract syntax that we will present is no doubt closer to C than to JVM. One reason is that what we are building is a *C compiler*, and match with the target language is a secondary consideration. Another, more general reason is that C is a higher-level language and JVM which means, among other things, that C makes more semantic distinctions. In general, the abstract syntax of a translation system must reflect all semantic distinctions that can be made in the languages involved, and then it is a good idea to start with looking at what the most distinctive language needs.

3.1 Statements

Statements in C may involve variables, expressions, and other statements. The following `cat` judgements of GF define the syntactic categories that are needed to construct statements

```
cat
  Stm ;
  Typ ;
  Exp Typ ;
  Var Typ ;
```

The type `Typ` is the type of C's datatypes. The type of expressions is a dependent type, since it has a nonempty context, indicating that `Exp` takes a `Typ` as argument. The rules for `Exp` will thus be rules to construct well-typed expressions of a given type. `Var` is the type of variables, of a given type, that get bound in C's variable declarations.

Let us start with the simplest kind of statements: declarations and assignments. The following `fun` rules define their abstract syntax:

```
fun
  Decl   : (A : Typ) -> (Var A -> Stm) -> Stm ;
  Assign : (A : Typ) -> Var A -> Exp A -> Stm -> Stm ;
```

The `Decl` function captures the rule that a variable must be declared before it can be used or assigned to: its second argument is a *continuation*, which is the sequence of statements that depend on (= may refer to) the declared variable. The `Assign` function uses dependent types to control that a variable is always assigned a value of proper type.

We will treat all statements, except `return`s, in terms of continuations. A sequence of statements (which always has the type `Stm`) thus always ends in a `return`, or, abruptly, in an empty statement, `End`. Here are rules for some other statement forms:

```
    While  : Exp TInt -> Stm -> Stm -> Stm ;
    IfElse : Exp TInt -> Stm -> Stm -> Stm -> Stm ;
    Block  : Stm -> Stm -> Stm ;
    Return : (A : Typ) -> Exp A -> Stm ;
    End    : Stm ;
```

Here is an example of a piece of code and its abstract syntax.

```
  int x ;      Decl (TNum TInt) (\x ->
  x = 5 ;        Assign (TNum TInt) x (EInt 5) (
  return x ;       Return (TNum TInt) (EVar (TNum TInt) x)))
```

The details of expression and type syntax will be explained in the next section.

Our binding syntax is more liberal than C's in two ways. First, lambda calculus permits overshadowing previous bindings by new ones, e.g. to write `\x -> (\x -> f x)`. The corresponding overshadowing of declarations is not legal in C, within one and the same block. Secondly, we allow declarations anywhere in a block, not just in the beginning. The second deviation would be easy to mend, whereas the first one is inherent to the method of higher-order abstract syntax.

3.2 Types and expressions

Our fragment of C has two types: integers and floats. Many operators of C are overloaded so that they can be used for both of these types, as well as for some other numeric types—but not for e.g. arrays and structures. We capture this distinction by a notion reminiscent of *type classes*: we introduce a special category of numeric types, and a coercion of numeric types into types in general.

```
  cat
    NumTyp ;
  fun
    TInt, TFloat : NumTyp ;
    TNum : NumTyp -> Typ ;
```

Well-typed expressions are built from constants, from variables, and by means of binary operations.

```
    EVar   : (A : Typ) -> Var A -> Exp A ;
    EInt   : Int -> Exp (TNum TInt) ;
    EFloat : Int -> Int -> Exp (TNum TFloat) ;
    ELt    : (n : NumTyp) -> let Ex = Exp (TNum n) in
               Ex -> Ex -> Exp (TNum TInt) ;
    EAdd, EMul, ESub : (n : NumTyp) -> let Ex = Exp (TNum n) in
               Ex -> Ex -> Ex ;
```

Notice that the category `Var` has no constructors, but its expressions are only created by variable bindings in higher-order abstract syntax. Notice also that GF has a built-in type `Int` of integer literals, but floats are constructed by hand.

Yet another expression form are function calls. To this end, we need a notion of (user-defined) functions and argument lists. The type of functions depends on an argument type list and a value type. Expression lists are formed to match type lists.

```
cat
  ListTyp ;
  Fun ListTyp Typ ;
  ListExp ListTyp ;

fun
  EAppNil : (V : Typ) -> Fun NilTyp V -> Exp V ;
  EApp    : (AS : ListTyp) -> (V : Typ) ->
              Fun AS V -> ListExp AS -> Exp V ;

  NilTyp  : ListTyp ;
  ConsTyp : Typ -> ListTyp -> ListTyp ;

  OneExp  : (A : Typ) -> Exp A -> ListExp (ConsTyp A NilTyp) ;
  ConsExp : (A : Typ) -> (AS : ListTyp) ->
              Exp A -> ListExp AS -> ListExp (ConsExp A AS) ;
```

The separation between zero-element applications and other applications is a concession to the concrete syntax of C: it in this way that we can control the use of commas so that they appear between arguments (`(x,y,z)`) but not after the last argument (`(x,y,z,)`). The compositionality of linearization (Section 4 below) forbids case analysis on the length of the lists.

3.3 Functions

On the top level, a program is a sequence of functions. Each function may refer to functions defined earlier in the program. The idea to express the binding of function symbols with higher-order abstract syntax is analogous to the binding of variables in statements, using a continuation. As with variables, the principal way to build function symbols is as bound variables (in addition, there can be some built-in functions, unlike in the case of variables). The continuation of can be recursive, which we express by making the function body into a part of the continuation; the category `Rec` is the combination of a function body and the subsequent function definitions.

```
cat
  Program ;
  Rec ListTyp ;
fun
  Empty : Program ;
  Funct : (AS : ListTyp) -> (V : Typ) ->
            (Fun AS V -> Rec AS) -> Program ;
  FunctNil : (V : Typ) ->
               Stm -> (Fun NilTyp V -> Program) -> Program ;
```

For syntactic reasons similar to function application expressions in the previous section, we have distinguished between empty and non-empty argument lists.

The tricky problem with function definitions is that they involve two nested binding constructions: the outer binding of the function symbol and the inner binding of the function parameter lists. For the latter, we could use vectors of variables, in the same way as vectors of expressions are used to give arguments to functions. However, this would lead to the need of cumbersome projection functions when using the parameters in the function body. A more elegant solution is to use higher-order abstract syntax to build function bodies:

```
RecOne  : (A : Typ) ->
               (Var A -> Stm) -> Program -> Rec (ConsTyp A NilTyp) ;
RecCons : (A : Typ) -> (AS : ListTyp) ->
              (Var A -> Rec AS) -> Program -> Rec (ConsTyp A AS) ;
```

The end result is an abstract syntax whose relation to concrete syntax is somewhat remote. Here is an example of the code of a function and its abstract syntax:

```
                    let int = TNum TInt in
int fact            Funct (ConsTyp int NilTyp) int (\fact ->
      (int n) {                                       RecOne int (\n ->
  int f ;             Decl int (\f ->
  f = 1 ;             Assign int f (EInt 1) (
  while (1 < n) {     While (ELt int (EInt 1) (EVar int n)) (Block (
    f = n * f ;         Assign int f (EMul int (EVar int n) (EVar int f)) (
    n = n - 1 ;         Assign int n (ESub int (EVar int n) (EInt 1))
  }                     End))
  return f ;          (Return int (EVar int f))) End)))
} ;                   Empty)
```

3.4 The printf function

To give a valid type to the C function `printf` is one of the things that can be done with dependent types [2]. We have not defined `printf` in its full strength, partly because of the difficulties to compile it into JVM. But we use special cases of `printf` as statements, to be able to print values of different types.

```
Printf : (A : Typ) -> Exp A -> Stm -> Stm ;
```

4 The concrete syntax of C

A concrete syntax, for a given abstract syntax, consists of `lincat` judgements

$$\texttt{lincat}\ C = T$$

defining the *linearization types* T of each category C, and `lin` judgements

$$\texttt{lin}\ f = t$$

defining the *linearization functions* t of each function f in the abstract syntax. The linearization functions are checked to be well-typed with respect the `lincat` definitions, and the syntax of GF forces them to be *compositional* in the sense that the linearization of a complex tree is always a function of the linearizations of the subtrees. Schematically, if

$$f : A_1 \rightarrow \cdots \rightarrow A_n \rightarrow A$$

then

$$f^{o} : A_1^{o} \rightarrow \cdots \rightarrow A_n^{o} \rightarrow A^{o}$$

and the linearization algorithm is simply

$$(f\ a_1\ \ldots\ a_n)^{o} = f^{o}\ a_1^{o}\ \ldots\ a_n^{o}$$

using the ^{o} notation for both linearization types, linearization functions, and linearizations of trees.

Because of compositionality, no case analysis on expressions is possible in linearization rules. The values of linearization therefore have to carry information on how they are used in different situations. Therefore linearization types are generally record types instead of just the string type. The simplest record type that is used in GF is

`{s : Str}`

If the linearization type of a category is not explicitly given by a `lincat` judgement, this type is used by default.

With higher-order abstract syntax, a syntax tree can have variable-bindings in its constituents. The linearization of such a constituent is compositionally defined to be the record linearizing the body extended with fields for each of the variable symbols:

$$(\lambda x_0 \rightarrow \cdots \rightarrow \lambda x_n \rightarrow b)^{o} = b^{o} \ast\ast \{\$_0 = x_0^{o}; \ldots ; \$_n = x_n^{o}\}$$

Notice that the variable symbols can always be found because linearizable trees are in η-long normal form. Also notice that we are here using the ^{o} notation in yet another way, to denote the magic operation that converts variable symbols into strings.

4.1 Resource modules

Resource modules define auxiliary notions that can be used in concrete syntax. These notions include *parameter types* defined by `param` judgements

$$\texttt{param}\ P = C_1\,\Gamma_1 \mid \cdots \mid C_n\,\Gamma_n$$

and *operations* defined by `oper` judgements

$$\texttt{oper}\ f : T = t$$

These judgements are similar to datatype and function definitions in functional programming, with the restriction that parameter types must be finite and operations may not be recursive. It is due to these restrictions that we can always derive a parsing algorithm from a set of linearization rules.

In addition to types defined in `param` judgements, initial segments of natural numbers, `Ints n`, can be used as parameter types. This is the most important parameter type we use in the syntax of C, to represent precedence.

The following string operations are useful in almost all grammars. They are actually included in a GF `Prelude`, but are here defined from scratch to make the code shown in the Appendices complete.

```
oper
  SS    : Type = {s : Str} ;
  ss    : Str -> SS = \s -> {s = s} ;
  cc2   : (_,_ : SS) -> SS = \x,y -> ss (x.s ++ y.s) ;
  paren : Str -> Str = \str -> "(" ++ str ++ ")" ;
```

4.2 Precedence

We want to be able to recognize and generate one and the same expression with or without parentheses, depending on whether its precedence level is lower or higher than expected. For instance, a sum used as an operand of multiplication must be in parentheses. We capture this by defining a parameter type of precedence levels. Five levels are enough for the present fragment of C, so we use the enumeration type of integers from 0 to 4 to define the *inherent precedence level* of an expression

```
oper
  Prec    : PType = Predef.Ints 4 ;
  PrecExp : Type = {s : Str ; p : Prec} ;
```

in a resource module (see Appendix D), and

```
lincat Exp = PrecExp ;
```

in the concrete syntax of C itself.

To build an expression that has a certain inherent precedence level, we use the operation

```
mkPrec : Prec -> Str -> PrecExp = \p,s -> {s = s ; p = p} ;
```

To use an expression of a given inherent level at some expected level, we define a function that says that, if the inherent level is lower than the expected level, parentheses are required.

```
usePrec : PrecExp -> Prec -> Str = \x,p ->
  ifThenStr
    (less x.p p)
    (paren x.s)
    x.s ;
```

(The code shown in Appendix D is at the moment more cumbersome, due to holes in the support for integer arithmetic in GF.)

With the help of `mkPrec` and `usePrec`, we can now define the main high-level operations that are used in the concrete syntax itself—constants (highest level), non-associative infixes, and left associative infixes:

```
constant : Str -> PrecExp = mkPrec 4 ;

infixN : Prec -> Str -> (_,_ : PrecExp) -> PrecExp = \p,f,x,y ->
  mkPrec p (usePrec x (nextPrec p) ++ f ++ usePrec y (nextPrec p)) ;
infixL : Prec -> Str -> (_,_ : PrecExp) -> PrecExp = \p,f,x,y ->
  mkPrec p (usePrec x p ++ f ++ usePrec y (nextPrec p)) ;
```

(The code in Appendix D adds to this an associativity parameter, which is redundant in GF, but which we use to instruct the Happy parser generator.)

4.3 Expressions

With the machinery introduced, the linearization rules of expressions are simple and concise:

```
    EVar   _ x   = constant x.s ;
    EInt     n   = constant n.s ;
    EFloat a b = constant (a.s ++ "." ++ b.s) ;
    EMul _       = infixL 3 "*" ;
    EAdd _       = infixL 2 "+" ;
    ESub _       = infixL 2 "-" ;
    ELt _        = infixN 1 "<" ;

    EAppNil val f = constant (f.s ++ paren []) ;
    EApp args val f exps = constant (f.s ++ paren exps.s) ;
```

4.4 Types

Types are expressed in two different ways: in declarations, we have `int` and `float`, but as formatting arguments to `printf`, we have `"%d"` and `"%f"`, with the quotes belonging to the names. The simplest solution in GF is to linearize types to records with two string fields.

```
  lincat
    Typ, NumTyp = {s,s2 : Str} ;
  lin
    TInt    = {s = "int" ; s2 = "\"%d\""} ;
    TFloat  = {s = "float" ; s2 = "\"%f\""} ;
```

4.5 Statements

Statements in C have the simplest linearization type, `{s : Str}`. We use a handful of auxiliary operations to regulate the use of semicolons on a high level.

```
  oper
    continues : Str -> SS -> SS = \s,t -> ss (s ++ ";" ++ t.s) ;
    continue  : Str -> SS -> SS = \s,t -> ss (s ++ t.s) ;
    statement : Str -> SS       = \s   -> ss (s ++ ";");
```

As for declarations, which bind variables, we notice the projection `.$0` to refer to the bound variable. Also notice the use of the `s2` field of the type in `printf`.

```
  lin
    Decl  typ cont = continues (typ.s ++ cont.$0) cont ;
    Assign _ x exp = continues (x.s ++ "=" ++ exp.s) ;
    While exp loop = continue  ("while" ++ paren exp.s ++ loop.s) ;
    IfElse exp t f = continue  ("if" ++ paren exp.s ++ t.s ++ "else" ++ f.s) ;
    Block stm      = continue  ("{" ++ stm.s ++ "}") ;
    Printf t e     = continues ("printf" ++ paren (t.s2 ++ "," ++ e.s)) ;
    Return _ exp   = statement ("return" ++ exp.s) ;
    Returnv        = statement "return" ;
    End            = ss [] ;
```

4.6 Functions and programs

The category Rec of recursive function bodies with continuations has three components: the function body itself, the parameter list, and the program that follows. We express this by a linearization type that contains three strings:

```
  lincat Rec = {s,s2,s3 : Str} ;
```

The body construction rules accumulate the parameter list independently of the two other components:

```
  lin
    RecOne typ stm prg = stm ** {
      s2 = typ.s ++ stm.$0 ;
      s3 = prg.s
      } ;
    RecCons typ _ body prg = {
      s  = body.s ;
      s2 = typ.s ++ body.$0 ++ "," ++ body.s2 ;
      s3 = prg.s
      } ;
```

The top-level program construction rules rearrange the three components into a linear structure:

```
    FunctNil val stm cont = ss (
      val.s ++ cont.$0 ++ paren [] ++ "{" ++
      stm.s ++ "}" ++ ";" ++ cont.s) ;
    Funct args val rec = ss (
      val.s ++ rec.$0 ++ paren rec.s2 ++ "{" ++
      rec.s ++ "}" ++ ";" ++ rec.s3) ;
```

5 The concrete syntax of JVM

JVM syntax is, linguistically, more straightforward than the syntax of C, and could even be defined by a regular expression. However, the JVM syntax that our compiler generates does not comprise full JVM, but only the fragment that corresponds to well-formed C programs.

The JVM syntax we use is a symbolic variant of the Jasmin assembler [11]. The main deviation from Jasmin are variable addresses, as described in Section 5.1. The other deviations have to do with spacing: the normal unlexer of GF puts spaces between constituents, whereas in JVM, type names are integral parts of instruction names. We indicate gluing uniformly by generating an underscore on the side from which the adjacent element is glued. Thus e.g. `i _load` becomes `iload`.

5.1 Symbolic JVM

What makes the translation from our abstract syntax to JVM tricky is that variables must be replaced by numeric addresses (relative to the frame pointer). Code generation must therefore maintain a symbol table that permits the lookup of variable addresses. As shown in the code in Appendix C, we do not treat symbol tables in linearization, but instead generated code in *Symbolic JVM*—that is, JVM with symbolic addresses. Therefore we need a postprocessor that resolves the symbolic addresses, shown in Appendix E.

To make the postprocessor straightforward, Symbolic JVM has special `alloc` instructions, which are not present in real JVM. Our compiler generates `alloc` instructions from variable declarations. The postprocessor comments out the `alloc` instructions, but we found it a good idea not to erase them completely, since they make the code more readable.

The following example shows how the three representations (C, Symbolic JVM, JVM) look like for a piece of code.

```
int x ;   alloc i x      ; x gets address 0
int y ;   alloc i y      ; y gets address 1
x = 5 ;   ldc 5          ldc 5
          i _store x     istore 0
y = x ;   i _load x      iload 0
          i _store y     istore 1
```

5.2 Labels and jumps

A problem related to variable addresses is the generation of fresh labels for jumps. We solve this in linearization by maintaining a growing label suffix as a field of the linearization of statements into instructions. The problem remains that statements on the same nesting level, e.g. the two branches of an `if-else` statement can use the same labels. Making them unique must be added to the post-processing pass. This is always possible, because labels are nested in a disciplined way, and jumps can never go to remote labels.

As it turned out laborious to thread the label counter to expressions, we decided to compile comparison expressions (`x < y`) into function calls, and provide the functions in a run-time library. This will no more work for the conjunction (`x && y`) and disjunction (`x || y`), if we want to keep their semantics lazy, since function calls are strict in their arguments.

5.3 How to restore code generation by linearization

Since postprocessing is needed, we have not quite achieved the goal of code generation as linearization—if linearization is understood in the sense of GF. In GF, linearization can only depend on parameters from finite parameter sets. Since the size of a symbol table can grow indefinitely, it is not possible to encode linearization with updates to and lookups from a symbol table, as is usual in code generation.

One attempt we made to achieve JVM linearization with numeric addresses was to alpha-convert abstract syntax syntax trees so that variables get indexed with integers that indicate their depths in the tree. This hack works in the present fragment of C because all variables need the same amount of memory (one word), but would break down if we added double-precision floats. Therefore we have used the less pure (from the point of view of code generation as linearization) method of symbolic addresses.

It would certainly be possible to generate variable addresses directly in the syntax trees by using dependent types; but this would clutter the abstract syntax in a way that is hard to motivate when we are in the business of describing the syntax of C. The abstract syntax would have to, so to say, anticipate all demands of the compiler's target languages.

5.4 Problems with the JVM bytecode verifier

An inherent restriction for linearization in GF is compositionality. This prevents optimizations during linearization by clever instruction selection, elimination of superfluous labels and jumps, etc. One such optimization, the removal of unreachable code (i.e. code after a `return` instruction) is actually required by the JVM byte code verifier. The solution is, again, to perform this optimization in postprocessing. What we currently do, however, is to be careful and write C programs so that they always end with a return statement in the outermost block.

Another problem related to `return` instructions is that both C and JVM programs have a designated `main` function. This function must have a certain type, which is different in C and JVM. In C, `main` returns an integer encoding what errors may have happend during execution. The JVM `main`, on the other hand, returns a `void`, i.e. no value at all. A `main` program returning an integer therefore provokes a JVM bytecode verifier error. The postprocessor could take care of this; but currently we just write programs with void `return`s in the `main` functions.

The parameter list of `main` is also different in C (empty list) and JVM (a string array `args`). We handle this problem with an *ad hoc* postprocessor rule.

Every function prelude in JVM must indicate the maximum space for local variables, and the maximum evaluation stack space (within the function's own stack frame). The locals limit is computed in linearization by maintaining a counter field. The stack limit is blindly set to 1000; it would be possible to set an accurate limit in the postprocessing phase.

6 Translation as linearization vs. transfer

Many of the problems we have encountered in code generation by linearization are familiar from translation systems for natural languages. For instance, to translate the English pronoun *you* to German, you have to choose between *du, ihr, Sie*; for Italian, there are four variants, and so on. To deal with this by linearization, all semantic distinctions made in any of the involved languages have to be present in the common abstract syntax. The usual solution to this problem is not a universal abstract syntax, but *transfer*: translation does not just linearize the same syntax trees to another language, but uses a noncompositional function that translates trees of one language into trees of another.

Using transfer in the back end is precisely what traditional compilers do. The transfer function in our case would be a noncompositional function from the abstract syntax of C to a different abstract syntax of JVM. The abstract syntax notation of GF permits definitions of functions, and the GF interpreter can be used for evaluating terms into normal form. Thus one could write the code generator just like in any functional language: by sending in an environment and a syntax tree, and returning a new environment with an instruction list:

```
fun
  transStm : Env -> Stm -> EnvInstr ;
def
  transStm env (Decl typ cont) = ...
  transStm env (While (ELt a b) stm cont) = ...
  transStm env (While exp stm cont) = ...
```

This would be cumbersome in practice, because GF does not have programming-language facilities like built-in lists and tuples, or monads. Moreover, the compiler could no longer be inverted into a decompiler, in the way true linearization can be inverted into a parser.

7 Parser generation

The whole GF part of the compiler (parser, type checker, Symbolic JVM generator) can be run in the GF interpreter. The weakest point of the resulting compiler, by current standards, is the parser. GF is a powerful grammar formalism, which needs a very general parser, taking care of ambiguities and other problems that are typical of natural languages but should be overcome in programming languages by design. The parser is moreover run in an interpreter that takes the grammar (in a suitably compiled form) as an argument.

Fortunately, it is easy to replace the generic, interpreting GF parser by a compiled LR(1) parser. GF supports the translation of a concrete syntax into the *Labelled BNF* (LBNF) format, which in turn can be translated to parser generator code (Happy, Bison, or JavaCUP), by the BNF Converter [3]. The parser we are therefore using in the compiler is a Haskell program generated by Happy [10].

We regard parser generation as a first step towards developing GF into a production-quality compiler compiler. The efficiency of the parser is not the only relevant thing. Another advantage of an LR(1) parser generator is that it performs an analysis on the grammar finding conflicts, and provides a debugger. It may be difficult for a human to predict how a context-free grammar performs at parsing; it is much more difficult to do this for a grammar written in the abstract way that GF permits (cf. the example in Appendix B).

The current version of the C grammar is ambiguous. GF's own parser returns all alternatives, whereas the parser generated by Happy rules out some of them by its normal conflict handling policy. This means, in practice, that extra brackets are sometimes needed to group staments together.

7.1 Another notation for higher-order abstract syntax

Describing variable bindings with higher-order abstract syntax is sometimes considered unintuitive. Let us consider the declaration rule of C (without type dependencies for simplicity):

```
fun Decl : Typ -> (Var -> Stm) -> Stm ;
lin Decl typ stm = {s = typ.s ++ stm.$0 ++ ";" ++stm.s} ;
```

Compare this with a corresponding LBNF rule (also using a continuation):

```
Decl. Stm ::= Typ Ident ";" Stm ;
```

To explain bindings attached to this rule, one can say, in natural language, that the identifier gets bound in the statement that follows. This means that syntax trees formed by this rule do not have the form `(Decl typ x stm)`, but the form `(Decl typ (\x -> stm))`.

One way to formalize the informal binding rules stated beside BNF rules is to use *profiles*: data structures describing the way in which the logical arguments of the syntax tree are represented by the linearized form. The declaration rule can be written using a profile notation as follows:

```
Decl [1,(2)3]. Stm ::= Typ Ident ";" Stm ;
```

When compiling GF grammars into LBNF, we were forced to enrich LBNF by a (more general) profile notation (cf. [15], Section 3.3). This suggested at the same time that profiles could provide a user-fiendly notation for higher-order abstract syntax avoiding the explicit use of lambda calculus.

8 Using the compiler

Our compiler is invoked, of course, by the command `gfcc`. It produces a JVM `.class` file, by running the Jasmin bytecode assembler [11] on a Jasmin (`.j`) file:

```
% gfcc factorial.c
> > wrote file factorial.j
Generated: factorial.class
```

The Jasmin code is produced by a postprocessor, written in Haskell (Appendix E), from the Symbolic JVM format that is produced by linearization. The reasons why actual Jasmin is not generated by linearization are explained in Section 5.1 above.

In addition to the batch compiler, GF provides an interactive syntax editor, in which C programs can be constructed by stepwise refinements, local changes, etc. [6]. The user of the editor can work simultaneously on all languages involved. In our case, this means that changes can be done both to the C code and to the JVM code, and they are automatically carried over from one language to the other.

9 Related work

The theoretical ideas behind our compiler experiment are familiar from various sources. Single-source language and compiler definitions can be built using attribute grammars [7]. The use of dependent types in combination with higher-order abstract syntax has been studied in various logical frameworks [4, 9, 14]. The addition of linearization rules to type-theoretical abstract syntax is studied in [12], which also compares the method with attribute grammars.

The idea of using a common abstract syntax for different languages was clearly exposed by Landin [8]. The view of code generation as linearization is a central aspect of the classic compiler textbook by Aho, Sethi, and Ullman [1]. The use of one and the same grammar both for parsing and linearization is a guiding principle of unification-based linguistic grammar formalisms [13]. Interactive editors derived from grammars have been developed in various programming and proof assistants [17, 5, 9].

Even though the different ideas are well-known, we have not seen them used together to construct a complete compiler. In our view, putting these ideas together is an attractive approach to compiling, since a compiler written in this way is completely declarative, hence concise, and therefore easy to modify and extend. What is more, if a new language construct is added, the GF type checker verifies that the addition is propagated to all components of the compiler. As the implementation is declarative, it is also self-documenting, since a human-readable grammar defines the syntax and static semantics that is actually used in the implementation.

10 Conclusion

The `gfcc` compiler translates a representative fragment of C to JVM, and growing the fragment does not necessarily pose any new kinds of problems. Using higher-order abstract syntax and dependent types to describe the abstract syntax of C works fine, and defining the concrete syntax of C on top of this using GF linearization machinery is possible. To build a parser that is more efficient than GF's generic one, GF offers code generation for standard parser tools.

One result of the experiment is the beginning of a library for dealing with typical programming language structures such as precedence. This library is exploited in the parser generator, which maps certain parameters used into GF grammars into precedence directives in labelled BNF grammars.

The most serious difficulty with JVM code generation by linearization is to maintain a symbol table mapping variables to addresses. The solution we have chosen is to generate Symbolic JVM, that is, JVM with symbolic addresses, and translate the symbolic addresses to (relative) memory locations by a postprocessor.

Since the postprocessor works uniformly for the whole Symbolic JVM, building a new compiler to generate JVM should now be possible by just writing GF grammars. The most immediate idea for developing GF as a compiler tool is to define a similar symbolic format for an intermediate language, which uses three-operand code and virtual registers.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1988.

[2] L. Augustsson. Cayenne—a language with dependent types. In *Proc. of ICFP'98.* ACM Press, September 1998.

[3] M. Forsberg and A. Ranta. BNF Converter Homepage. `http://www.cs.chalmers.se/~markus/BNFC/`, 2002–2004.

[4] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *JACM*, 40(1):143–184, 1993.

[5] G. Kahn, B. Lang, B. Mélèse, and E. Morcos. Metal: a formalism to specify formalisms. *Science of Computer Programming*, 3:151–188, 1983.

[6] J. Khegai, B. Nordström, and A. Ranta. Multilingual Syntax Editing in GF. In A. Gelbukh, editor, *Intelligent Text Processing and Computational Linguistics (CICLing-2003), Mexico City, February 2003*, volume 2588 of *LNCS*, pages 453–464. Springer-Verlag, 2003.

[7] D. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–145, 1968.

[8] P. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, 1966.

[9] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, LNCS 806, pages 213–237. Springer, 1994.

[10] S. Marlow. Happy, The Parser Generator for Haskell, 2001. `http://www.haskell.org/happy/`.

[11] J. Meyer and T. Downing. *Java Virtual Machine.* O'Reilly, 1997.

[12] P. Mäenpää. Semantic BNF. In E. Gimenez and C. Paulin-Mohring, editors, *Types for Proofs and Programs, TYPES'96*, volume 1512 of *LNCS*, pages 196–215. Springer-Verlag, 1998.

[13] F. Pereira and S. Shieber. *Prolog and Natural-Language Analysis.* CSLI, Stanford, 1987.

[14] F. Pfenning. The Twelf Project. `http://www-2.cs.cmu.edu/~twelf`, 2002.

[15] A. Ranta. Grammatical Framework: A Type-Theoretical Grammar Formalism. *The Journal of Functional Programming*, 14(2):145–189, 2004.

[16] A. Ranta, K. Angelov, and T. Hallgren. Grammatical Framework Homepage. `http://grammaticalframework.org`, 2000–2009.

[17] T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, 1981.

Appendix A: The abstract syntax

```
abstract Imper = PredefAbs ** {
  cat
    Program ;
    Rec ListTyp ;
    Typ ;
    NumTyp ;
    ListTyp ;
    Fun ListTyp Typ ;
    Body ListTyp ;
    Stm ;
    Exp Typ ;
    Var Typ ;
    ListExp ListTyp ;

  fun
    Empty : Program ;
    Funct : (AS : ListTyp) -> (V : Typ) -> (Fun AS V -> Rec AS) -> Program ;
    FunctNil : (V : Typ) -> Stm -> (Fun NilTyp V -> Program) -> Program ;
    RecOne  : (A : Typ) -> (Var A -> Stm) -> Program -> Rec (ConsTyp A NilTyp) ;
    RecCons : (A : Typ) -> (AS : ListTyp) ->
                 (Var A -> Rec AS) -> Program -> Rec (ConsTyp A AS) ;

    Decl    : (A : Typ) -> (Var A -> Stm) -> Stm ;
    Assign  : (A : Typ) -> Var A -> Exp A -> Stm -> Stm ;
    While   : Exp (TNum TInt) -> Stm -> Stm -> Stm ;
    IfElse  : Exp (TNum TInt) -> Stm -> Stm -> Stm -> Stm ;
    Block   : Stm -> Stm -> Stm ;
    Printf  : (A : Typ) -> Exp A -> Stm -> Stm ;
    Return  : (A : Typ) -> Exp A -> Stm ;
    Returnv : Stm ;
    End     : Stm ;

    EVar   : (A : Typ) -> Var A -> Exp A ;
    EInt   : Int -> Exp (TNum TInt) ;
    EFloat : Int -> Int -> Exp (TNum TFloat) ;
    ELt    : (n : NumTyp) -> let Ex = Exp (TNum n) in Ex -> Ex -> Exp (TNum TInt) ;
    EAdd, EMul, ESub : (n : NumTyp) -> let Ex = Exp (TNum n) in Ex -> Ex -> Ex ;
    EAppNil : (V : Typ) -> Fun NilTyp V -> Exp V ;
    EApp    : (AS : ListTyp) -> (V : Typ) -> Fun AS V -> ListExp AS -> Exp V ;

    TNum   : NumTyp -> Typ ;
    TInt, TFloat : NumTyp ;
    NilTyp  : ListTyp ;
    ConsTyp : Typ -> ListTyp -> ListTyp ;
    OneExp  : (A : Typ) -> Exp A -> ListExp (ConsTyp A NilTyp) ;
    ConsExp : (A : Typ) -> (AS : ListTyp) ->
                Exp A -> ListExp AS -> ListExp (ConsTyp A AS) ;
}
```

Appendix B: The concrete syntax of C

```
concrete ImperC of Imper = open ResImper in {
  flags lexer=codevars ; unlexer=code ; startcat=Program ;

  lincat
    Exp = PrecExp ;
    Typ, NumTyp = {s,s2 : Str} ;
    Rec = {s,s2,s3 : Str} ;
  lin
    Empty = ss [] ;
    FunctNil val stm cont = ss (
      val.s ++ cont.$0 ++ paren [] ++ "{" ++ stm.s ++ "}" ++ ";" ++ cont.s) ;
    Funct args val rec = ss (
      val.s ++ rec.$0 ++ paren rec.s2 ++ "{" ++ rec.s ++ "}" ++ ";" ++ rec.s3) ;
    RecOne typ stm prg = stm ** {
      s2 = typ.s ++ stm.$0 ;
      s3 = prg.s
      } ;
    RecCons typ _ body prg = {
      s  = body.s ;
      s2 = typ.s ++ body.$0 ++ "," ++ body.s2 ;
      s3 = prg.s
      } ;

    Decl  typ cont = continues (typ.s ++ cont.$0) cont ;
    Assign _ x exp = continues (x.s ++ "=" ++ exp.s) ;
    While exp loop = continue  ("while" ++ paren exp.s ++ loop.s) ;
    IfElse exp t f = continue  ("if" ++ paren exp.s ++ t.s ++ "else" ++ f.s) ;
    Block stm      = continue  ("{" ++ stm.s ++ "}") ;
    Printf t e     = continues ("printf" ++ paren (t.s2 ++ "," ++ e.s)) ;
    Return _ exp   = statement ("return" ++ exp.s) ;
    Returnv        = statement "return" ;
    End            = ss [] ;

    EVar  _ x  = constant x.s ;
    EInt    n  = constant n.s ;
    EFloat a b = constant (a.s ++ "." ++ b.s) ;
    EMul _     = infixL 3 "*" ;
    EAdd _     = infixL 2 "+" ;
    ESub _     = infixL 2 "-" ;
    ELt _      = infixN 1 "<" ;
    EAppNil val f = constant (f.s ++ paren []) ;
    EApp args val f exps = constant (f.s ++ paren exps.s) ;

    TNum t  = t ;
    TInt = {s = "int" ; s2 = "\"%d\""} ; TFloat  = {s = "float" ; s2 = "\"%f\""} ;
    NilTyp  = ss [] ; ConsTyp = cc2 ;
    OneExp _ e = e ; ConsExp _ _ e es = ss (e.s ++ "," ++ es.s) ;
}
```

Appendix C: The concrete syntax of JVM

```
concrete ImperJVM of Imper = open ResImper in {
  flags lexer=codevars ; unlexer=code ; startcat=Program ;

  lincat
    Rec = {s,s2,s3 : Str} ; -- code, storage for locals, continuation
    Stm = Instr ;

  lin
    Empty = ss [] ;
    FunctNil val stm cont = ss (
      ".method" ++ "public" ++ "static" ++ cont.$0 ++ paren [] ++ val.s ++ ";" ++
      ".limit" ++ "locals" ++ stm.s2 ++ ";" ++
      ".limit" ++ "stack" ++ "1000" ++ ";" ++
      stm.s ++
      ".end" ++ "method" ++ ";" ++ ";" ++
      cont.s
      ) ;
    Funct args val rec = ss (
      ".method" ++ "public" ++ "static" ++ rec.$0 ++ paren args.s ++ val.s ++ ";" ++
      ".limit"  ++ "locals" ++ rec.s2 ++ ";" ++
      ".limit"  ++ "stack"  ++ "1000" ++ ";" ++
      rec.s ++
      ".end" ++ "method" ++ ";" ++ ";" ++
      rec.s3
      ) ;

    RecOne typ stm prg = instrb typ.s (
      ["alloc"] ++ typ.s ++ stm.$0 ++ stm.s2) {s = stm.s ; s2 = stm.s2 ; s3 = prg.s};

    RecCons typ _ body prg = instrb typ.s (
      ["alloc"] ++ typ.s ++ body.$0 ++ body.s2)
         {s = body.s ; s2 = body.s2 ; s3 = prg.s};

    Decl  typ cont = instrb typ.s (
      ["alloc"] ++ typ.s ++ cont.$0
      ) cont ;
    Assign t x exp = instrc (exp.s ++ t.s ++ "_store" ++ x.s) ;
    While exp loop =
      let
        test = "TEST_" ++ loop.s2 ;
        end = "END_" ++ loop.s2
      in instrl (
        "label" ++ test ++ ";" ++
        exp.s ++
        "ifeq" ++ end ++ ";" ++
        loop.s ++
        "goto" ++ test ++ ";" ++
        "label" ++ end
        ) ;
    IfElse exp t f =
      let
        false = "FALSE_" ++ t.s2 ++ f.s2 ;
        true  = "TRUE_" ++ t.s2 ++ f.s2
      in instrl (
        exp.s ++
        "ifeq" ++ false ++ ";" ++
        t.s ++
        "goto" ++ true ++ ";" ++
        "label" ++ false ++ ";" ++
        f.s ++
        "label" ++ true
        ) ;
    Block stm      = instrc stm.s ;
    Printf t e     = instrc (e.s ++ "invokestatic" ++ t.s ++ "runtime/printf" ++ paren (t.s)
    Return t exp   = instr (exp.s ++ t.s ++ "_return") ;
    Returnv        = instr "return" ;
    End            = ss [] ** {s2,s3 = []} ;

    EVar  t x  = instr (t.s ++ "_load" ++ x.s) ;
    EInt    n  = instr ("ldc" ++ n.s) ;
    EFloat a b = instr ("ldc" ++ a.s ++ "." ++ b.s) ;
    EAdd       = binopt "_add" ;
    ESub       = binopt "_sub" ;
    EMul       = binopt "_mul" ;
    ELt t  = binop ("invokestatic" ++ t.s ++ "runtime/lt" ++ paren (t.s ++ t.s) ++ "i") ;
    EAppNil val f = instr (
      "invokestatic" ++ f.s ++ paren [] ++ val.s
      ) ;
    EApp args val f exps = instr (
      exps.s ++
      "invokestatic" ++ f.s ++ paren args.s ++ val.s
      ) ;

    TNum t = t ;
    TInt   = ss "i" ;
    TFloat = ss "f" ;
    NilTyp = ss [] ;
    ConsTyp = cc2 ;
    OneExp _ e = e ;
    ConsExp _ _ = cc2 ;
}
```

Appendix D: Auxiliary operations for concrete syntax

```
resource ResImper = open Predef in {

  -- precedence

  param PAssoc = PN | PL | PR ;

  oper
    Prec    : PType = Predef.Ints 4 ;
    PrecExp : Type = {s : Str ; p : Prec ; a : PAssoc} ;

    mkPrec : Prec -> PAssoc -> Str -> PrecExp = \p,a,f ->
      {s = f ; p = p ; a = a} ;

    usePrec : PrecExp -> Prec -> Str = \x,p ->
      case <<x.p,p> : Prec * Prec> of {
        <3,4> | <2,3> | <2,4> => paren x.s ;
        <1,1> | <1,0> | <0,0> => x.s ;
        <1,_> | <0,_>         => paren x.s ;
        _ => x.s
        } ;

    constant : Str -> PrecExp = mkPrec 4 PN ;

    infixN : Prec -> Str -> (_,_ : PrecExp) -> PrecExp = \p,f,x,y ->
      mkPrec p PN (usePrec x (nextPrec p) ++ f ++ usePrec y (nextPrec p)) ;
    infixL : Prec -> Str -> (_,_ : PrecExp) -> PrecExp = \p,f,x,y ->
      mkPrec p PL (usePrec x p ++ f ++ usePrec y (nextPrec p)) ;
    infixR : Prec -> Str -> (_,_ : PrecExp) -> PrecExp = \p,f,x,y ->
      mkPrec p PR (usePrec x (nextPrec p) ++ f ++ usePrec y p) ;

    nextPrec : Prec -> Prec = \p -> case <p : Prec> of {
      4 => 4 ;
      n => Predef.plus n 1
      } ;

  -- string operations

    SS    : Type = {s : Str} ;
    ss    : Str -> SS = \s -> {s = s} ;
    cc2   : (_,_ : SS) -> SS = \x,y -> ss (x.s ++ y.s) ;
    paren : Str -> Str = \str -> "(" ++ str ++ ")" ;

    continues : Str -> SS -> SS = \s,t -> ss (s ++ ";" ++ t.s) ;
    continue  : Str -> SS -> SS = \s,t -> ss (s ++ t.s) ;
    statement : Str -> SS       = \s   -> ss (s ++ ";");

  -- operations for JVM
```

```
    Instr  : Type = {s,s2,s3 : Str} ; -- code, variables, labels
    instr  : Str -> Instr = \s ->
      statement s ** {s2,s3 = []} ;
    instrc : Str -> Instr -> Instr = \s,i ->
      ss (s ++ ";" ++ i.s) ** {s2 = i.s2 ; s3 = i.s3} ;
    instrl : Str -> Instr -> Instr = \s,i ->
      ss (s ++ ";" ++ i.s) ** {s2 = i.s2 ; s3 = "L" ++ i.s3} ;
    instrb : Str -> Str -> Instr -> Instr = \v,s,i ->
      ss (s ++ ";" ++ i.s) ** {s2 = v ++ i.s2 ; s3 = i.s3} ;
    binop  : Str -> SS -> SS -> SS = \op, x, y ->
      ss (x.s ++ y.s ++ op ++ ";") ;
    binopt : Str -> SS -> SS -> SS -> SS = \op, t ->
      binop (t.s ++ op) ;
}
```

Appendix E: Translation of Symbolic JVM to Jasmin

```
module Main where
import Char
import System

main :: IO ()
main = do
  jvm:src:_ <- getArgs
  s <- readFile jvm
  let cls = takeWhile (/='.') src
  let obj = cls ++ ".j"
  writeFile  obj $ boilerplate cls
  appendFile obj $ mkJVM cls s
  putStrLn $ "wrote file " ++ obj

mkJVM :: String -> String -> String
mkJVM cls = unlines . reverse . fst . foldl trans ([],([],0)) . lines where
  trans (code,(env,v)) s = case words s of
    ".method":p:s:f:ns
        | f == "main" -> (".method public static main([Ljava/lang/String;)V":code,([],1))
        | otherwise  -> (unwords [".method",p,s, f ++ typesig ns] : code,([],0))
    "alloc":t:x:_  -> (("; " ++ s):code, ((x,v):env, v + size t))
    ".limit":"locals":ns -> chCode (".limit locals " ++ show (length ns))
    "invokestatic":t:f:ns | take 8 f == "runtime/" ->
          chCode $ "invokestatic " ++ "runtime/" ++ t ++ drop 8 f ++ typesig ns
    "invokestatic":f:ns  -> chCode $ "invokestatic " ++ cls ++ "/" ++ f ++ typesig ns
    "alloc":ns           -> chCode $ "; " ++ s
    t:('_':instr):[";"]  -> chCode $ t ++ instr
    t:('_':instr):x:_    -> chCode $ t ++ instr ++ " " ++ look x
    "goto":ns            -> chCode $ "goto " ++ label ns
    "ifeq":ns            -> chCode $ "ifeq " ++ label ns
    "label":ns           -> chCode $ label ns ++ ":"
    ";":[] -> chCode ""
    _ -> chCode s
   where
     chCode c = (c:code,(env,v))
     look x   = maybe (error $ x ++ show env) show $ lookup x env
     typesig  = init . map toUpper . concat
     label    = init . concat
     size t   = case t of
       "d" -> 2
       _ -> 1

boilerplate :: String -> String
boilerplate cls = unlines [
  ".class public " ++ cls, ".super java/lang/Object",
  ".method public <init>()V","aload_0",
  "invokenonvirtual java/lang/Object/<init>()V","return",
  ".end method"]
```