A formal definition based design of the translation phase of a CHILL compiler

Author: C.A. Middelburg

Date: october 1980

Cover note

This paper has been prepared for the
"CHILL Implementors/Users Meeting"
held in Copenhagen, 20 - 24 october 1980.

© Copyright 1980 Dr. Neher Laboratory, Netherlands PTT
P.O.Box 421, 2260 AK Leidschendam, The Netherlands
tel. +31 70 755080, telex 31236 dnl nl

Abstract

This paper describes the design of the translation phase of the CHILL compiler developed at the Dr. Neher-Laboratory of the Netherlands PTT. Special attention is paid to the role of the formal definition of CHILL in the design process.

Contents

1.0 INTRODUCTION 1

2.0 ENVIRONMENT 1

3.0 OVERVIEW OF THE COMPILER 2

4.0 THE DESIGN OF THE TRANSLATION PHASE 3
- 4.1 SUMMARY OF THE FORMAL DEFINITION 4
- 4.2 A MORE CONCRETE SEMANTICS 5
  - 4.2.1 Sequential Elaboration 5
  - 4.2.2 Linear Storage 6
- 4.3 AN ABSTRACT TRANSLATING ALGORITHM 7
  - 4.3.1 The Description of Environment and State 8
  - 4.3.2 The Translation Functions 9
  - 4.3.3 The Abstract Program Tree 10
- 4.4 A CONCRETE TRANSLATING ALGORITHM 12
  - 4.4.1 The Machine Independent Translation Phase 12
  - 4.4.2 The Machine Dependent Translation Phase 13

1.0 INTRODUCTION

In april 1976 a CHILL compiler construction project was initiated at the Dr. Neher-Laboratory; the research and development laboratory of the Netherlands Postal and Telecommunications Services. The actual compiler construction was taken up in september 1977. Completion of the compiler is to be expected at the begin of 1981. By that time the manpower invested will be about 13 man-years.

The aims of the project are twofold:

1. to gain knowledge and experience in the fields of high-level languages for SPC programming, and of compiler construction;

2. to evaluate the proposed language CHILL, and to contribute to its improvement.

These aims have the following consequences for the design of the compiler:

1. the most recent advances in compiler construction technology must be taken into account;

2. efficiency of the compiler is far less important than flexibility with respect to changes in the language definition.

The main part of this paper outlines the design of the translation phase of the compiler. Besides it describes briefly the characteristics and structure of the compiler. It must be stressed that for clearness' sake a lot of details have been left out intensionally; e.g. concurrent processing and mapped mode objects are completely left out of consideration.

2.0 ENVIRONMENT

The compiler will run on a DECsystem-10 and will generate symbolic assembly code for PDP11 processors, acceptable to the MACRO11 assemblers available on these processors and the DECsystem-10.

The compiler and all of the auxiliary programs (such as a parser generator) are written entirely in Pascal. The Pascal compiler used is an adapted version of the one made at the University of Hamburg.

3.0 OVERVIEW OF THE COMPILER

The compilation process can be conceptually divided into a recognition (analysis) phase and a translation (synthesis) phase. For reasons of flexibility, we have decided to maintain this division also in the implementation.

The recognition phase has been implemented as a two-pass process. The main reason for this is that complete analysis of a CHILL program may not be done in one traversal of the program, since the syntactical and/or semantical analysis of some constructs may depend on the applying definition or declaration of an identifier, which in the absence of a "define before use" rule may be unknown up to nearly the end of the traversal.

The first recognition pass includes:

1. lexical analysis;

2. (context-free and context-sensitive) syntactical analysis as far as possible independent of the applying definitions/declarations of identifiers;

3. complete syntactical and semantical analysis of definition and declaration statements, including evaluation of literal expressions within these constructs;

4. transformation of the program text, in order to be able to complete the analysis in the next pass.

The second recognition pass includes:

complete syntactical and semantical analysis; of course already treated definition and declaration statements are not analysed again.

Each of these passes is a syntax-driven, bottom-up recognizer. The actual parsing is done by interpreting an instruction code for a push-down transducer. This parser code is generated, from the syntax for either pass, by a semi-automatic LALR(1) parser generator of our own construction.

In the first pass, the semantical analysis of definition and declaration statements may not be done completely "on the fly", since the semantical analysis may depend on other definitions/declarations which are either not yet semantically analysed or incompletely analysed too. Therefore, this pass is further divided into an "on the fly" part and a "completion" part.

In the second pass, semantical analysis is done completely "on the fly" except for the evaluation of constant expressions in synonym definitions. At the time that the implementation of this pass was taken up only the draft language definition (known as the "Blue Document") [1] was available. This document was considered unsuitable as a basis for implementation. For this reason we decided to make an attribute grammar formulation of CHILL. Later

we found the greater part of the revised language definition (known as the "Brown Document") [2] may be considered a less formal version of our attribute grammar!

The translation phase has been implemented as an one-pass process; thus no global optimization is performed. In contrast, much attention is paid to local optimizations. The translation pass is a "piecewise coding" translator, driven by "sequential interpretation" rules. The actual translation is directed by interpreting an instruction code for a push-down transducer. This "translator" code is generated, from the interpretation rules, by a generator of our own construction. For reasons of flexibility and portability, this (machine independent) part of the translator is completely isolated from the (machine dependent) coding part.

4.0 THE DESIGN OF THE TRANSLATION PHASE

As will be put forward in the following outline, the design of the translation phase has been based on the formal definition of CHILL [3]. This definition, which is not (yet) part of the official language definition, provides a "denotational semantics" for CHILL expressed in the so-colloquially-called META-IV notation.

The considerations to proceed this way are:

1. The description of CHILL as offered in the Brown Document is less suitable as a basis for the design of the translation phase; it is conveniently arranged to get a clear view of the syntax and the static semantics, by which the rules with respect to the dynamic semantics become too much scattered. On the contrary, the formal definition gives a clear "global" as well as "local" description of the dynamic semantics.

2. The approach of Denotational Semantics is at present one of the most precise methods to obtain a constructive definition of the dynamic semantics of CHILL and as such pre-eminently suitable to obtain a reliable implementation. So its use is in full agreement with the aims of the project.

In the sequel it is assumed that the reader is familiar with the approach of Denotational Semantics and with the META-IV notation. He who is not is referred to [4] and [5] respectively.

4.1 SUMMARY OF THE FORMAL DEFINITION

The "context free" syntax of CHILL is specified by defining syntactic domains which are mathematical abstractions of the concrete syntactic categories. Thus, abstraction has been made from representational details.

The static "context" conditions of CHILL are specified by defining is-well-formed functions which map syntactic constructs to truth values, designating whether or not the context conditions are met.

The "dynamic" semantics of CHILL is specified by defining semantic domains which model the concepts by means of which the meaning of the various syntactic constructs is explained, and by defining elaboration functions which map syntactic constructs to their mathematical "meaning" (i.e. the semantic objects which they denote).

The meaning of the various syntactic constructs is basically explained by means of the following concepts:dictionary (static context), environment (dynamic context) and state.
Dictionaries are finite maps which primarily map each identifier to an object which reflects its current static properties:

DICT = (Id $\underset{m}{\rightarrow}$ Descr) $\underline{\cup}$ ...

Descr = Locd | Vald | Procd | ...

Environments are finite maps which primarily map each identifier to its current denotation, i.e. the semantic object (location, value, procedure, etc.) it currently denotes:

ENV = (Id $\underset{m}{\rightarrow}$ Den) $\underline{\cup}$ ...

Den = LOC | VAL | PROC | ...

The principal component of a state is a storage. Storages are finite maps which map each allocated location to its current contents, i.e. the value it currently contains:

STATE = ($\underline{\text{STG}}$ $\underset{m}{\rightarrow}$ STG) $\underline{\cup}$ ...

STG = LOC $\underset{m}{\rightarrow}$ VAL

The basic elaboration functions have the following functionalities:

int-Action: Action $\simeq$> ((DICT ENV) $\simeq$> (STATE $\simeq$> STATE))

eval-ValExpr: ValExpr $\simeq$> ((DICT ENV) $\simeq$> (STATE $\simeq$> (STATE VAL)))

eval-LocExpr: LocExpr $\simeq$> ((DICT ENV) $\simeq$> (STATE $\simeq$> (STATE LOC)))

4.2 A MORE CONCRETE SEMANTICS

The task of the translation phase is to produce appropriate machine code from a CHILL program. Here, the machine code is considered appropriate if the effect of its execution is that given by the semantics for the program. The denotational semantics however, is too abstract to direct the actual translation. Consequently we need an equivalent but more concrete semantics, which is, like present-day computers, based on sequential elaboration (execution) and a simple linear storage (memory).

4.2.1 Sequential Elaboration

All conditional, iterative and recursive elaboration which must be fixed dynamically is realized by means of a simple jump mechanism. For that purpose META-IV has been extended with two constructs:

```
hop label [ unless expr ]

stmt1 ; entry label ; stmt2
```

This "hop mechanism" can be considered a simplified exit mechanism; suffice it to say that the constructs introduced can be given a mathematical meaning expressed in lambda notation, i.e. they are sugared forms of lambda expressions just like true META-IV constructs.

To outline the approach it is shown how the conditional elaboration of an If-action is realized.

The (simplified) "conditional" elaboration function:

```
int-If(mk-If(e,al1,al2))(dict)(env)=

 if eval-ValExpr(e)(dict)(env)

  then int-Actlst(al1)(dict)(env)

  else int-Actlst(al2)(dict)(env)
```

The "sequential" elaboration function:

```
s-int-If(mk-If(e,al1,al2))(dict)(env)=

   let lelse : new-lbl(),

       lout  : new-lbl();

 ((let b : s-eval-ValExpr(e)(dict)(env);

   execute(mk-Jmpf(b,lelse))(dict)(env);

   s-int-Actlst(al1)(dict)(env);
```

```
        execute(mk-Jmp(lout))(dict)(env);
        entry lelse; I);
        s-int-Actlst(al2)(dict)(env);
        entry lout; I)
    where
      execute(op)(dict)(env)=
      (cases op:
       ...
       mk-Jmpf(b,lbl) -> hop lbl unless b,
       mk-Jmp(lbl)    -> hop lbl          ,
       ...
      )
```

Rigorous arguments for the correctness of such sequential elaborations are relatively simple. However formal proofs involve "desugaring" of the notation.

4.2.2 Linear Storage

Linear storage is considered a list of "directly addressable memory units", e.g. bytes. Therefore it is obvious to model locations by natural numbers which are indices (in machine terms: addresses) of the storage and to model values by lists of directly addressable memory units which are sublists of the storage. To support those aspects of storage which can be fixed statically, "ghost components" must be joined to these objects:

```
cSTG :: s-stg: Byte*  s-locs: cLOC-set

cLOC :: s-addr: Nat₀  s-md: Mode  [ BASE-LOC ]

cVAL :: s-bytes: Byte+  s-md: Mode
```

Functions can be defined to reconstruct the abstract storages, locations and values from their concrete counterparts:

```
retr-stg(mk-cSTG(cstg,clocs))=
 [retr-loc(cloc) -> retr-val(mk-cVAL(cstg[s-addr(cloc)],s-md(cloc))) |
  cloc ∈ clocs]

type: cSTG ~> STG
```

```
retr-loc(mk-cLOC(addr,md))=

(cases md:

 ...

 mk-ArrayMode(imd,emd) ->

   let mk-DiscrMode(lb,ub)=imd,

       step=size(emd) `in

   <retr-loc(mk-cLOC(addr+(i-lb)*step,emd)) | lb <= i <= ub > ,

 ...

)

type: cLOC ~> LOC


retr-val(mk-cVAL(bytes,md))=

 ...

type: cVAL ~> VAL
```

The operations on the concrete representations of storages , locations and values which model the operations on the abstract storages, locations and values can be specified implicitly in terms of these "retrieve functions".

4.3 AN ABSTRACT TRANSLATING ALGORITHM

The essential realization idea is that the translation of a program is carried out by simulating the (concrete) elaboration of the program as follows:

1. When the elaboration indicates that the environment or state should be altered, then code to perform the alteration is emitted and a description of the environment and/or state is updated.

2. When the elaboration indicates that a location or value should be determined, then code to determine the location or value is emitted and a description of that location or value is produced.

4.3.1 The Description of Environment and State

Storage descriptions are rather complex since they must supply the needs of such aspects of storage management as dynamic storage allocation for declared locations (including formal parameters) and anonymous locations (temporaries), accessing of locations and calling of procedures. A storage description contains a component which gives for the current procedure and its statically embracing procedures a description which contains a reference to the access description of the pointer whose (changeable) value will be the "base address" of the Dynamic "data Storage/save Area" (DSA) allocated for the locally declared locations and formal parameters of the most recent invocation of the procedure concerned, and the sizes of the components of the DSA.
Another component maps references to access descriptions to the actual access descriptions. This indirection step is introduced to support the optimization of address computation.

```
Stated = ( STG -m-> Stgd ) U ...

Stgd :: (Proclvl -m-> DsaDescr )  ( AccRef -m-> AccDescr )

DsaDescr :: s-bp-acc: AccRef  s-sz-fpl: Nat0 s-sz-lvl: Nat0 ...

AccDescr :: Addrspec  Sizespec  [ TMP ]  ...

Addrspec = Regno | StaticAddr | BasedAddr | IndexedAddr

Regno :: Nat0

StaticAddr :: Nat0

BasedAddr :: s-bp-acc: AccRef  s-disp: Intg

IndexedAddr :: s-aggr-acc: AccRef s-ind-acc: AccRef s-step: Nat
```

The addressing structure introduced here, on the one hand fits properly with the CHILL location structure and on the other hand takes into account the architecture of most present-day computers and familiar methods of storage management.

The environment descriptions are maps which map each identifier to a description containing all the information pertaining to the determination of its changeable denotation in procedure invocations. It is obvious, that location and value descriptions both contain a reference to an access description as principal component. Procedure descriptions contain a label indicating the start of the code generated for the procedure concerned and a reference to the access description of the pointer whose changeable value will be the base address of the DSA associated with the most recent invocation of the statically direct embracing procedure, i.e. it is a description of a "closure" (see [2]).

```
Envd = ( Id -m-> cDenDescr) ∪ ...

cDenDescr = cLocd | cVald | cProcd | ...

cLocd :: AccRef  Mode  [ BASE-LOC ]  ...

cVald :: ( AccRef | IMM )  Mode  [ ConstVal ]  ...

cProcd :: s-lbl: ProcLbl  s-epa-bp-acc: AccRef  ...

...
```

Functions can be defined to extract a concrete storage, etc. from its description and the actual machine state comprising of a machine storage, a register set, etc. .
The simulation of the operations on concrete storages, etc. can be specified implicitly in terms of these functions.

4.3.2 The Translation Functions

For each elaboration function a corresponding translation function need to be defined. Given the sequential elaboration functions, this is a straightforward process.

To outline this process, it is shown how the translation of If-actions (the sequential elaboration of which is already shown) is realized.

```
transl-If(mk-If(e,al1,al2))(dict)=

 let lelse : new-lbl(),

     lout  : new-lbl();

 let vd : transl-ValExpr(e)(dict);

 simulate(JMPF,<vd,lelse>)(dict);

 transl-Actlst(al1)(dict);

 simulate(JMP,<lout>)(dict);

 emit-lbl(lelse);

 transl-Actlst(al2)(dict);

 emit-lbl(lout)

type: If ≃> ( DICT ≃> ( tSTATE ≃> tSTATE ))
```

where

$$tSTATE = (ENV \xrightarrow[m]{} Envd) \cup (STATE \xrightarrow[m]{} Stated) \cup$$
$$(LBLS \xrightarrow[m]{} Lbl\text{-}set) \cup (CODE \xrightarrow[m]{} mCode)$$

A great deal of confidence in the correctness of the translation is provided by proceeding in the way outlined sofar.

4.3.3 The Abstract Program Tree

Because an abstract program (as defined in the formal definition) reflects the syntax of a program (it may be considered a parse tree), it is not well suited to translate from. Better suited is an object which differs from an abstract program in the following ways:

1. all of its structure is relevant from a semantical point of view,
2. the meaning of each of its components is independent of its static context.

Such an object is an "Abstract Program Tree" (APT):

```
Apt :: Op  [ Apt ]

Op = Module ¦ ... ¦ If ¦ ... ¦ Boolor ¦ Bitwor ¦ Union ¦ ...

Module :: Opnd

...

If :: Opnd  Opnd  Opnd

...

Boolor :: Opnd  Opnd

...

Opnd = PrimOpnd ¦ Op ¦ OpndSeq

PrimOpnd = DefObj ¦ ConstVal ¦ FldSel ¦ AuxilObj

DefObj :: DefObjRef  Descr

AuxilObj = DecTbl ¦ RcvCaseTbl ¦ ...

OpndSeq :: Opnd*
```

It must be noted that the APT can be produced by the recognition phase without increasing its complexity.

In the following, the rather straightforward translation of an APT is sketched.

```
transl-Apt(mk-Apt(op,apt))=

 transl-Op(op,nil);

 if apt = nil then I else transl-Apt(apt)

type: Apt ≃> ( tSTATE ≃> tSTATE )


transl-Op(op,tmode)=

(cases op:

  ...

  mk-Add(opnd1,opnd2) ->

    let vd1 : transl-Opnd(opnd1,nil);

    let vd2 : transl-Opnd(opnd2,nil);

    simul-op(ADD,<vd1,vd2>) ,

  ...

)

type: ( Op  [QUOT] ) ≃> ( tSTATE ≃> ( tSTATE  [ cDenDescr ] ))


transl-Opnd(opnd,tmode)=

(cases opnd:

  mk-PrimOpnd(prim) -> simul-primopnd(prim,tmode),

  mk-Op(op)         -> transl-Op(op,tmode),

  mk-OpndSeq(opnds) ->

    let t-opnds(opnds',tmode')=

     (if opnds' = <>

        then I

        else transl-Opnd(hd opnds',tmode');t-opnds(tl opnds',tmode'))

    in t-opnds(opnds,tmode)

)
```

type: (Opnd [QUOT]) ≃> (tSTATE ≃> (tSTATE [cDenDescr]))

The environment describing component of the "translator state" (tSTATE) must be changed as follows:

Envd = (DefObjRef $\underset{m}{\rightarrow}$ cDenDescr) $\underline{\cup}$ ...

4.4 A CONCRETE TRANSLATING ALGORITHM

The main part of the functions shown, only directs the translation: production of machine code and maintenance of descriptions is only done by the functions simul-op and simul-primopnd. In other words the translation process can be divided into a machine independent phase which produces code for a hypothetical (sequential) machine especially designed for CHILL, and a machine dependent phase which simulates the execution of this code. In the sequel, the instructions for this hypothetical machine will be called CML instructions (CML for "CHILL Machine Language").

4.4.1 The Machine Independent Translation Phase

The translation of the different operations have a common pattern:

1. The operands of an operation are translated "from left to right".
2. CML instructions are generated (if applicable to the operation concerned):
   - before the translation of the first operand (prefix encounter),
   - between the translation of two operands (infix encounters),
   - after the translation of the last operand (postfix encounter).
3. The arguments of each CML instruction generated always correspond to the last translated operands.
4. If certain operands of an operation have to be translated in an "abnormal translation mode", tranlation mode transition always occurs with an encounter of the operation.

Thus this translation can be performed by interpretation of a set of "translation rules" (loosely called "interpretation rules"), one for each (APT) operation; e.g.:

```
IF -> prefix :     .newlbl lelse,lout

      infix1 :     jmpf lelse

      infix2 :     jmp  lout;

                   entry lelse

      postfix:     entry lout .
```

A linear representation of the APT is used. This representation is based on knowledge for each operation about the encounters at which CML instructions must be generated and/or the translation mode must be changed. The resulting representation of a subtree of which the root denotes an operation Op, has one of the following forms:

prefix form:
"Op" <opnd repr> { "INFIX" <opnd repr> }* "POSTFIX"

infix form:
{ <opnd repr> }+
"Op" <opnd repr> { "INFIX" <opnd repr> }* "POSTFIX"

postfix form:
{ <opnd repr> }* "Op"

e.g.
- "IF" 'valexpr' "INFIX" 'actlst1' "INFIX" 'actlst2' "POSTFIX"
- 'valexpr' "BOOLOR" 'valexpr' "POSTFIX"
- 'locexpr or valexpr' 'valexpr' "INDEXING"

4.4.2 The Machine Dependent Translation Phase

The simulation of the execution of CML instructions involves the following:

1. making arguments machine addressable;
2. allocating and freeing machine storage space and registers;
3. generating symbolic assembly code;
4. creating, updating and deleting descriptions;
5. analysing special cases.

The original idea was to perform this simulation by interpretation of a set of "coding rules", one for each CML operation, expressed in a special "Interpretive Coding Language" (ICL); a language with commands to perform the tasks mentioned above. However, the special case analysis needed to produce reasonable efficient machine code requires a high-level conditional construct such as a "decision table" case conditional, interpretation of which was considered unfeasible.
The final approach was to create an explicit specification of the simulation of each operation, i.e. its coding rule, expressed in an extended META-IV notation and to realize these coding rules in Pascal.

To illustrate the special case analysis involved, it is shown how the execution of the Add instruction is simulated.

```
simul-add(src1,src2)=

(if (knd(src1)=CONST ∧ val(src1) ∈ {-1,0,1}) ∨
    (knd(src2)=CONST ∧ val(src2) ∈ {-1,0,1})
 then  ...
 else
  (cases knd(src1) , knd(src2):
    (CONST),(CONST) ->                      cplt-eval(ADD,<src1,src2>),
    (CONST),(REGTMP,STKTMP) ->              g-gen2(ADD,src1,src2),
    (CONST),(VAR) ->                        let tmp : g-push(src2);
                                            g-gen2(ADD,src1,tmp),
    (REGTMP,STKTMP),(REGTMP) ->             g-gen2(ADD,src1,src2),
    (REGTMP,STKTMP),(CONST,STKTMP,VAR) ->   g-gen2(ADD,src2,src1),
    (VAR),(CONST) ->                        let tmp : g-push(src1);
                                            g-gen2(ADD,src2,tmp),
    (VAR),(REGTMP,STKTMP,VAR) ->            let tmp : g-load(src2);
                                            g-gen2(ADD,src1,tmp)
  )
)
```

Given such simulation functions, their realization in Pascal is a rather straightforward process.

References

[1] The HLL Team of Specialists
"Proposal for a Recommandation for a C.C.I.T.T. High Level programming Language (2nd edition)"
The "Blue Document" of CCITT Study Group XI
CCITT Secretariat, may 1977

[2] CCITT Study Group XI
"CHILL Language Definition (Brown Document)"
CCITT Secretariat, may 1980

[3] D. Bjorner, P.L. Haff
"CHILL Formal Definition"
Danish Datamatics Centre, august 1980

[4] J.E. Stoy
"Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory"
MIT Press, june 1977

[5] D. Bjorner, C.B. Jones
"The Vienna Development Method: The Meta-Language"
Springer Lecture Notes in Computer Science, number 61, january 1978