Implementing Haskell: Language Implementation as a Tool Building Exercise *†

Diomidis Spinellis
Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ
e-mail: dds@doc.ic.ac.uk

Abstract

Although a number of tool boxes for compiler construction exist, the language implementation task can often be made easier by building specialised tools. A prototype Haskell system was implemented within a four month period using such an approach. The system is currently used as a front end for a transputer array, Haskell implementation. In this article we describe the tool building aspect of the implementation process. The little language tools, tree processing function generators and error message management routines developed are described. Although the tools are specific to this implementation, this mode of work can be worthwhile in a number of cases such as the implementation of novel languages, the targeting of unconventional architectures or, the experimentation with new implementation techniques. The lessons learned during this process are summarised and the specialised tool approach is evaluated.

1 Introduction

A great number of tools [19] and many complete tool boxes [13, 32] are available today to aid the language implementor. During the process of implementing a system supporting the Haskell language [17], we found that considerable advances in productivity and reliability can be obtained by building specialised little tools catering for specific language-related problems. The system was implemented in a four month period by a three person team and consists of approximately 15000 lines of code. A modified version of the system is currently used as a front end for the FAST project: "Functional Programming for Arrays of Transputers" [12]. The development was made possible by extensive use of software tools. Many of the standard and user-contributed tools available on the Unix operating system were used. In addition a number of custom tools were designed and implemented. These tools were needed because special features of the Haskell language — such as the *layout* syntax rule and type classes — made the use of complete compiler construction tool boxes (such as [13, 32]) infeasible. The new tools formed an integral part of the project development.

We believe that building highly specialised tools can, under certain circumstances, be a worthwhile approach. There are many cases where the implementation and of a custom tool can be preferable to the use of an existing one. The potential benefits from using a custom-implemented tool are:

Applicability: Many of the tools cater for mainstream approaches, so they may not be applicable when implementing unconventional languages, targeting novel architectures or, experimenting with new implementation techniques.

Efficiency: The narrow domain of a specific implementation, can make the approach used by a specialised tool more efficient, than the solution provided by a general purpose one.

Conciseness: Equivalently, knowledge of the application domain may render the full set of specifications required by a general purpose tool unnecessary. Thus the specialised tool input language can be more concise.

*Software: Concepts & Tools 14:37–48, 1993.

It has been argued [24] that the rôle of the software engineer is to support the development process by organising it and providing the suitable tools. The importance of the tool builder in a development team is also described in [5]. Software tools have been described in [21] and their importance to language development in [19, 20].

The tools developed, can be divided in three categories:

- compilers for special-purpose little languages,
- tree handling procedure generators and,
- error message management.

2 Project Overview

Before we describe the individual tools in detail we present an overview of the system and its development environment. The project specification was to develop an experimental implementation of the Haskell programming language.

> "Haskell is a general-purpose, purely functional programming language exhibiting many of the recent innovations in functional (as well as other) programming language research, including higher order functions, lazy evaluation, static polymorphic typing, user-defined data types, pattern matching and list comprehensions. It is also a very complete language in that is has a module facility, a well-defined functional I/O system, and a rich set of primitive data types including lists, arrays, arbitrary and fixed precision integers, and floating point numbers." [16, p. 381]

The front end of the system consists of a command line interpreter. From that interface, the user can develop a program using the interpreter or compile it into an executable file. The interpreter and the compiler are built into the same system and share the same front end.

Figure 1 presents an overview of the system structure. A description of the functional language implementation techniques and an explanation of the terminology used can be found in [11, 27], however familiarity with functional programming technology is not needed for understanding the rest of the article. The system can be roughly divided into the following parts:

Lexical analysis: The Haskell source is scanned converting the text into tokens [29].

Parsing: The stream of tokens from the lexical analyser is converted into a parse tree [29].

Type checking: The parse tree is type-checked and augmented with type information [34].

Conversion to supercombinators: The Haskell parse tree is semantically checked and translated into enriched λ-calculus with pattern abstractions, then into pure λ-calculus with *letrec* expressions and finally, using λ-lifting, into supercombinators [15].

Interpreter: An eval-apply interpreter is applied on the sugared λ-calculus tree [29]. Currently, the garbage collection scheme described in [4], is used.

G-code generation: Translation of supercombinator expressions to abstract G-machine code [15].

Native code generation: Translation of abstract G-machine code into native machine code [29].

The executable version of the system is created by linking together 23 modules written by hand in ANSI C with six modules automatically generated as follows:

- the parser written in *yacc*,
- the character class type description written in *ct* described in section 3.1,
- the Haskell primitives written in *prim* described in section 3.2,
- the tree copy functions automatically generated by *makecp* as described in section 4.1,
- the tree print functions automatically generated by *makepr* as described in section 4.2 and,

- the list inversion functions automatically generated by *makeinv* as described in section 4.3.

The source code involved in the generation of the system, also includes a back-end that translates the G-machine code into the native assembly language. This is automatically generated from *md2c*, a machine description file compiler described in section 3.3. The error handling routines of the system, use an error message database generated by scanning all the C code as explained in section 5. The breakup of the total coding effort, between hand-generated code, automatically-generated code, and tool implementation is illustrated in fig. 2.

3 Three Little Languages and their Compilers

In the following sections we will describe three little languages that were used for implementing parts of the Haskell system. Examples of little languages developed in the same spirit can be found in [1, 2, 3].

3.1 Character Class Compiler

A number of tools are available for supporting the generation of lexical analysers. Some of them are *lex* [25], *flex* [26], and *gperf* [28]. The lexical analysis of the Haskell source is a non-trivial problem. Although the initial implementation of the Haskell scanner was based on *flex*, after some time we found, that the regular expressions were getting too complicated trying to cater for a number of special cases and parser tie-ins. Haskell has a number of characteristics, such as the layout rule and the definable precedence and associativity of operators that made the use of a hand-crafted scanner preferable. The hand crafted scanner, needed a set of functions that would efficiently classify characters as belonging to a particular set.

The character class compiler compiles *ct*, a special language used for creating such functions. These functions are analogous to the *ctype* functions of the C language library. Their single parameter is a character and they return TRUE if that character belongs to a given character class. An excerpt from a source code file in this language is shown bellow:

```
# Hexadecimal digit
H isxdigit              [a-fA-F0-9]

# Symbol start
S issymstart            [!#$%&*+./<=>?@\\^|~]
```

The language used to specify those functions is based on regular expressions. The language source consists of blank lines and comment lines starting with a # sign which are copied to the output source and lines containing three blank separated fields which form the function description. The three fields are:

Class identifier: A single character used as a constant for identifying the character class within the source file.

Function name: The function name that will be generated for testing membership to that character class.

Regular expression: A single-character matching regular expression, that describes the characters matched by the function. The regular expression can contain individual characters, character ranges, optionally complemented with respect to the target character set.

The language is compiled into C by reading all the regular expressions and creating a look-up table for each class. The look-up table is in the form of an array: the position equal to the ordinal value of the character in the look-up table will contain a value that is TRUE if the character belongs to the character class represented by the look-up table, and FALSE otherwise. The look-up tables for all classes are packed in a vertical manner in the form of an integer array as illustrated in fig. 3. Each element of the array is treated as a bit-vector, whose individual bits are 0 or 1, depending on the classes that character belongs to. The number of different character classes that can be accommodated by this approach depends on the bit size of the integer representation (32 in our case).

The array is accessed by a C macro with the name of the classification function. It works by *binary-anding* the appropriate array element with the bit-vector offset for that character class. An example of the automatically generated resulting file is given below:

```
/* Hexadecimal digit */
/* [a-fA-F0-9] */
```

```
#define ct_H 2
#define isxdigit(c) (ctype[(c)] & ct_H)
[...]
/* Symbol start */
/* [!#$%&*+./<=>?@\\^|~] */
#define ct_S 32
#define issymstart(c) (ctype[(c)] & ct_S)
[...]
int ctype[256] = {
0,                                                    /* 0 */
[...]
        ct_H | ct_R | ct_D | ct_G | ct_O | ct_T,            /* 3 */
        ct_H | ct_R | ct_D | ct_G | ct_O | ct_T,            /* 4 */
[...]
        ct_Y | ct_M | ct_G | ct_T | ct_E,                   /* @ */
        ct_H | ct_U | ct_R | ct_G | ct_T | ct_E,            /* A */
        ct_H | ct_U | ct_R | ct_G | ct_T | ct_E,            /* B */
```

The implementation described is quite efficient. Characters can be classified in about four machine instructions. An alternative approach using the `strchr` C library function would be around 15 times more expensive. As the lexical analysis is based around these functions the resulting efficiency improvement was significant.

3.2 Compiler for Haskell Primitives

Haskell can be defined using an extended version of the λ-calculus and a set of primitive operations. The primitives are low level Haskell functions that express basic notions of arithmetic or the operating environment. Primitives with a similar structure can be found in a number of interpretative or abstract-machine-based language implementations such as Prolog and Lisp. An example of a primitive is `primPlusFloat` which adds two floating point numbers. These primitives were described in *prim*, a special language which was then automatically translated into C.

All primitives have some common characteristics. Each primitive needs to:

- isolate its arguments, passed in the form of a linked list,
- evaluate the arguments into weak head normal form,
- check that the arguments passed are of the correct type (this is done in the debug version of the program to flag potential programmer errors),
- create a new node of the appropriate type,
- fill the appropriate field of the node with the result and,
- return that node.

The system, according to the initial specification, would be based around 23 primitives. More primitives would be needed if some operations were to be performed in a more efficient manner. In order to avoid repetitiveness and errors in the primitive implementation, a compiler for Haskell primitives was implemented. The small language providing the interface between C and Haskell contains features of both languages. It also uses some conventions such as the `$` pseudo-variable found in *yacc* [18]. The primitive description file can contain comments starting with the # character and blank lines. Code between {% and %} pairs is literally included in the resulting C output. Its purpose is to allow for the specification of include files, global variables, data structures and functions.

The user needs to specify a map between the Haskell types, their C representations, the C enumeration constants used to represent them in an expression and the union field they belong to. Each item of the map starts on a new line with a keyword `%type`. Map elements are separated by double colons. For example the map for fixed precision integer values is the following:

```
%type  Int      : int        : ex_int       : i
```

This means that a Haskell value of type `Int` is stored in the structure union field `u.i` with the structure `kind` field set to `ex_int`. A C variable of type `int` can store such a value.

After the map is given, the user can define the primitives. Primitives are defined by a line starting with the keyword `primitive`, followed by the name of the Haskell primitive and its Haskell type signature. A C block follows the primitive declaration. Within the block the pseudo-variables `$1`, `$2` etc. are substituted by the appropriate union fields of the actual arguments, while the pseudo-variable `$$` stands for the correctly initialised result tree node. The definition for the `primEqInt` primitive might look as follows:

```
primitive primEqInt :: Int -> Int -> Bool
{
       $$ = ($1 == $2);
}
```

This compiles into the following C code:

```
static struct s_expression *
primEqInt(struct s_expresslist *ppargs)
{
    struct s_expression *ppresult = xmalloc(sizeof(*ppresult));
    struct s_expression *pparg0;
    struct s_expression *pparg1;

    pparg0 = eval(ppargs->expr, NULL);
    assert(pparg0->kind == ex_int);
    pparg1 = eval(ppargs->next->expr, NULL);
    assert(pparg1->kind == ex_int);
    ppresult->kind = ex_int;
#line 83 "prim.pr"
{
    ppresult->u.i = (pparg0->u.i == pparg1->u.i);
}
    return ppresult;
}
```

The compiler for Haskell primitives automatically creates a header file with the C function declarations. The compiler was implemented in the Perl programming language. The associative arrays, variable substitution within strings and extended regular expressions of Perl significantly eased the implementation task. A description of the implementation is provided in section 6.

3.3 Machine Description Compiler

Declarative languages are usually compiled into instructions for an abstract machine [27, 37, 35]. The abstract machine instructions can then be either interpreted by an abstract machine interpreter (often written in assembly language), or compiled into native machine code. Our Haskell implementation generated assembly code for the abstract G-machine using a straightforward process described in [27, p. 293–366]. The generation of native machine code from G-code was then a simple matter of efficient macro expansion.

The machine description compiler creates an executable file that translates the intermediate G-machine assembly representation of the program, into native assembly language. The process of translating G-code into assembly code can be specified by using the concept of a *machine description file*. That file contains a mapping from G-instructions to native machine instruction sequences. A separate program, the *machine description compiler*, compiles the machine description file into an executable program that converts G-code into assembly. This approach has the following advantages:

- The developer need only focus on the assembly mapping of the G-instructions when writing the machine description file. Tasks such as the lexical analysis and parsing of the G-instructions do not become part of the problem.
- A machine description file is smaller and easier to write than a complete translator. Ports of the system to different architectures can be achieved by simply rewriting the machine description file.

- Changing the format of the G-code (e.g. converting it into a binary stream for efficiency reasons, or even directly connecting the two processes) will not invalidate the machine description files and the effort put into them. Simply a new machine description file compiler needs to be written.

The format of the machine description file was designed to be the following:

Comment lines are blank lines, or lines starting with the # character.

Header inclusion starts with the symbol `%%{`. All code from that point up to the matching `%}` is copied verbatim to the assembly file. The purpose of this section is to include assembler constant definitions, jump tables, macros etc.

Assembly comment definition is given by the sequence `%%comment`. The character following the word comment is taken to be the assembler comment character. When code for the translator is generated all comments in the assembly file will be removed. This allows the user to put arbitrary comments inside assembly sequences without affecting the efficiency of the translation process (one should remember that a particular instruction sequence can be repeated many times on the output). The comment facility of the definition language is not used, because it is quite probable that the comment character of the definition language (the # sign) will serve some other purpose in some assembler.

G instruction definitions are introduced by the `%keyword` sequence, where `keyword` is the name of a G-instruction. An optional list of formal parameters can be given. The text starting at the following line up to the first `%}` will be generated for that instruction by the translator. Any formal parameters will be substituted, during the process of translating G-code to native assembly language, by the actual parameters following the G-instruction. Any local assembly comments are removed at compile time.

An example from the 68000 machine description file is given bellow:

```
# Remove n elements (below the TOS element) from stack
%slide number
        movl    a5@,a5@(4*number)               | Copy TOS to new location
        addl    #-5 * number, a5                | Update ep
%}
```

The "machine description" to "G-code to assembly translator", compiler was implemented as a script written in Perl. The first implementation of the translator generated a *lex* [25] file that had rules for the translation process. However the file took excessively long to compile with *lex*, and the translators produced were very slow. A speedup of a factor of seven was achieved by replacing the *lex* output by a C program based on a perfect hash function. One might argue that the machine description file could be implemented by a macro processor such as *m4* [22]. This is true, however the macro processor would interpret the machine description file at run-time with a resulting loss of efficiency, whereas our approach compiled the machine description into executable code. An analogous approach is used in the project GNU C compiler [30].

4 Tools for Operating on Trees

Most of the system parts from the parser onwards, receive as input and produce as output a tree. Functions were needed that could copy, print and reverse trees or parts of them. A tool which provides this functionality is *ast* [14]. The difference between the *ast* approach and ours (other than tool scope) is that *ast* requires the specifications to be written using a special formalism (abstract syntax), whereas the tools implemented in our project are based on a subset of C type declarations.

Significant care was taken to define the trees in a uniform manner in order to facilitate the automatic generation of functions that would operate on them. An example of a tree node definition is given bellow:

```
/*
 * Declaration kinds
 */
enum e_decl {
        ed_sign,                                /* Type signature */
        ed_simpledef,                           /* Simple definition */
```

```
        ed_guarddef                             /* Guarded definition */
};

/*
 * Declarations
 */
struct s_decllist {
        struct s_decllist *next;                /* Next declaration */
        short line;                             /* Line number */
        short file;                             /* File number */
        enum e_decl kind;                       /* Declaration kind */
        union {
        struct {                                /* Type signature */
                struct s_classlist *context;    /* Class context */
                struct s_idlist *vars;          /* Variables */
                struct s_type *type;            /* Type */
        } s;
        struct {                                /* Simple definition */
                struct s_lhs *lhs;              /* Left hand side */
                struct s_expr *expr;            /* Expression */
        } i;
        struct {                                /* Guarded definition */
                struct s_lhs *lhs;              /* Left hand side */
                struct s_guardexprlist *gexprs; /* Guarded expressions */
        } g;
        } u;
};
```

The following naming and structuring conventions were followed:

- Structure tag names start with `s_`, enumeration tag names start with `e_`.
- No `typedef` type definitions are used.
- Unions that are members of structures are named `u`.
- For every union an enumerated type is defined, that is used to specify which element of the union is valid. The elements of the union and the enumerated type fields are defined in the same order. If two enumerations correspond to the same union member a blank line is left in the union definition. Enumerations without corresponding unions are placed at the end of the definition.
- Structures containing unions must possess a variable named `kind` which specifies which member of the union is valid.
- Linked lists are linked together with a pointer placed in a variable named `next`.

These rules may seem numerous, arbitrary and fascist. In practice however they did not come into our way and made the creation of tools that operated on trees easy, as the tools could be based on lexical hints rather than parsing the whole file (parsing was implemented using regular expressions). In the following sections we describe the tools developed.

4.1 Tree Copy

Some parts of the system required a copy of a part of the syntax tree, instead of the original tree, in order to perform various transformations on it. A small tool was developed that processed the C tree definition file and created a C source code file with functions to copy each structure. Unions are handled by a `switch` statement on the `kind` variable and pointers to structures by recursively invoking the appropriate copying function. An example of the code generated is given bellow:

```
struct s_decllist *
cp_s_decllist(struct s_decllist *m)
{
        struct s_decllist *new;

        if (!m)
                return (struct s_decllist *)NULL;
        new = (struct s_decllist *)xmalloc(sizeof(struct s_decllist));
        *new = *m;
        new->next = cp_s_decllist(new->next);
        switch (m->kind) {
        case ed_sign:
                new->u.s.context = cp_s_classlist(new->u.s.context);
                new->u.s.vars = cp_s_idlist(new->u.s.vars);
                new->u.s.type = cp_s_type(new->u.s.type);
                break;
        case ed_simpledef:
[...]
```

The tool generated 23 different copy functions corresponding to the different node types of the parse tree.

4.2 Tree Printing

A similar tool was developed in order to create functions for printing out the contents of a tree; an operation needed mainly for debugging purposes e.g. to verify the parser operation. The structure was traversed in a similar manner as in the tree copy program. A global variable was used to keep track of the indentation depth and produce a readable printout. This tool was developed at an early stage of the project and proved very useful as the tree structure was repeatedly modified in quest for the optimum one. Had separate debugging print-out routines been written, they would have needed modification, whereas in this case only a re-run of the tool was necessary.

4.3 Tree Reversal

The success of the previous two tools motivated us to use the same approach for solving another similar problem. The tree generated by the scanner was the reverse of the source code representation. This was a result of the LR parsing method, which added the parsed elements into the head of the various lists. This could be handled correctly at parse time in three distinct ways, each with with its own drawbacks:

Each list could contain a pointer to its last element: Would have made the tree representation more complicated.

For each element traverse the list to find its end: Would increase the complexity order of the algorithm.

Make the parser rules right recursive: Would place a burden on the fixed size parser internal stack.

The solution adopted was to create a tool that would recursively traverse all the tree, reversing the linked lists of the tree starting from their anchor using the method described in [31]. The structures representing linked lists could be recognised by searching for a structure element named `next`. The same element was then used for traversing the list, so this tool was easy to construct in a manner analogous to the two other tools.

4.4 General Comments on Tree Handling

The three tree handling programs were very similar. In retrospect we can envisage a meta-tool that gets a specification of an operation on a tree as input, and produces a tool that, operating on the representation of the tree, produces code that performs that operation. Other more complicated tools that operate on trees like *twig* [33] could be specified in this manner. This approach can provide functionality similar to the Ada *generics*. However, because the concept realisation is done at a meta-linguistic level, more powerful constructions are possible. A theoretically sound approach to this problem can be found in [8].

5 Error Message Management

Meaningful error messages are one of the most important aspects of a user interface. Cryptic error messages will confuse a novice user, while overly verbose messages can hide valuable details under their volume. Good error messages are very difficult to create as indicated in [7]. An approach often taken is to index the error messages by a code. Each error message is printed as a brief message accompanied with the index number. In the documentation, and quite often on-line, there is a list of error messages sorted by their index numbers together with more detailed information. The information provided should give possible reasons for the error which occurred, and suggest recovery actions [6, p. 309].

The features described above, present a serious logistic problem. Errors have to be numbered in a coherent way. Every time a new error message is added the indices and the documentation have to be updated. If on-line help is available then that needs to be kept synchronised. The best solution is for the documentation to be part of the program source as in [10]. An attractive solution to this specific problem, on which ours is based, is presented in [9].

Errors are kept in a database file. The cause of the error and the proposed recovery actions are integrated into the source code in the form of comments. A special program scans the source code and automatically creates the database. The mapping between the database and the actual error message is represented by the file name and line number in which the error was called. Special macros are used to take advantage of the C preprocessor ability to substitute the file name and line number for the identifiers: `__FILE__` and `__LINE__`. When an error function is called its parameters include the file name and the line number of the source file. The error reporting function, scans the error message database `errors.db` to find the error number assigned by the tool to that error, and prints it out together with the brief message. An example of the source code part (file `scan.c` line 640) where an error is reported is shown bellow:

```
error(__FILE__, __LINE__, , "Invalid decimal escape (\\%d)", v);
/*
 * An illegal decimal escape sequence was found.  The value of
 * the resulting character is higher than the maximum allowed.
 *
 * Give a decimal number from 0 to 255 and make sure that a
 * sequence of less than three decimal digits is not followed
 * by other digits producing a spurious result.
 */
```

A tool written in Perl scans all the source code and creates the following three files:

`Errors.db`: A file containing the source file and line number for every function reporting an error, together with the respective error number. It is used by the error reporting functions to associate and print an error number for a given error. This is achieved by calling the error report function with the file name and line number as parameters. An excerpt from this file is given bellow:

```
scan.c 640 2004
scan.c 660 2005
scan.c 681 2006
```

`Errors.txt`: Contains a sorted list of all errors codes, their respective messages, possible error causes and suggested corrective actions. The file is formatted in a uniform and visually attractive way using the report generator of Perl. This file is suitable for printing on a line printer or for display on a glass tty terminal. It is used by the help system of the interpreter to provide additional information when an error is encountered. An example from that file is given bellow:

```
2004: Invalid decimal escape (\<number>)

      Explanation: An illegal decimal escape sequence
      was found. The value of the resulting character is
      higher than the maximum allowed.

      Action: Give a decimal number from 0 to 255 and
```

```
make sure that a sequence of less than three
decimal digits is not followed by other digits
producing a spurious result.
```

`Errors.tex`: A file containing the error messages with suitable commands for printing by the LaTeX[23] document preparation system. This file was used in the production of the system documentation. An example from the system documentation printout is given bellow:

2004 Invalid decimal escape (*number*)

Reason An illegal decimal escape sequence was found. The value of the resulting character is higher than the maximum allowed.

Action Give a decimal number from 0 to 255 and make sure that a sequence of less than three decimal digits is not followed by other digits producing a spurious result.

In the two text files generated, the brief messages are suitably parameterised in order to hide the C `printf` output codes. For example the `printf` message:

```
"Invalid character \%c (\%d)\\n"
```

becomes "Invalid character *character* (*decimal number*)" in the LaTeXfile.

6 Implementation

In this section we will give a brief overview of the implementation of the tools described. All the tools were implemented in the Perl programming language [36]. Perl is a five year old language with an implementation freely available for Unix and MS-DOS-based machines.

The language, implemented as an interpreter, is optimised for scanning arbitrary text files, extracting information from them, and generating reports based on that information. It includes a rich set of operators, control flow statements, and built-in functions. It is block structured like Modula-2, with a C-like expression syntax. It supports associative arrays [1], lists, strings of arbitrary contents and length, powerful regular expressions, and has a built-in report generator.

Some basic Perl features needed to understand Perl code are the following: Perl variables are prefixed according to their contents: scalar variables (containing a single value) begin with a `$`, whole arrays with an `@`. Normal array elements are accessed using [ ] brackets, whereas associative are accessed using { } brackets. The index of the last element of a normal array can be found using the construct `$#array_name`. By default, many operations — like substitution (expressed as `s/pattern/new/`), and printing — are performed using as their value and result the special variable `$_`.

All the tools were implemented as a single Perl script. The typical structure was some initialisation code, followed by a loop on the input file, possibly followed by writing of stored results. Within the loop, the lines from the input file were matched against regular expressions. According to the match some operation was performed, possibly storing intermediate results in an associative array, or a list.

In order to make this description more concrete, in the following lines I will describe a bare-bones [2] implementation of the compiler for Haskell primitives as listed [3] in fig. 4. A reading knowledge of C and Unix-style regular expressions is assumed.

Line 1 opens the input and output files associating the descriptors `IN` and `OUT` with the respective file names. Line 2 starts a loop for all lines of the input file. The expression $<$ `IN` $>$ fetches the next line from the input file and implicitly assigns it to the variable `$_`. It returns `false` when the end of file is reached. Lines 3–41 are a big case selection depending on the regular expression that matches the input line. Line 3 matches type declarations: whitespace is removed in line 4, and in line 5, the declaration is split in five parts delimited with "`:`" which are assigned to the five variables on the left hand side. The variable `$haskell` contains the Haskell name for that type, while the other three contain the respective C type name, the union tag value (used for the type checking invariant) and the union member name. These are then assigned to three associative arrays in lines 6–8. For example, evaluating `$ctype{'Int'}` will return the value `int`, which is the C type associated with the Haskell `Int` type. The main body of Haskell primitives is dealt with in lines

[1] Associative arrays, are arrays indexed by arbitrary key values instead of integers.

[2] For brevity reasons, error handling code, and some special cases were omitted.

[3] The line numbers are provided for reference only.

9–41. After removing syntactic sugar (lines 10, 11), the name and the Haskell type signature are extracted in line 22 by splitting the implicit `$_` variable on the "`::`" pattern. Line 13 assigns every element of the type signature to a different element of the `@htypes` array, while line 14 adds at the end of the `@prims` array the name of the primitive. The arity of that primitive is stored in the associative array `%arity` in line 15. Lines 16–21 print the function header definition for the C function by interpolating variables within the print string, and lines 22–24 print the argument declarations. Code for evaluating each argument and verifying that it is of the correct type, is generated in lines 25–29. The `x` used in line 38 is the string repetition operator, used for accessing the appropriate element of the linked list by generating the appropriate number of pointer indirections through the `next` variable. Line 30 generates code to set the result type tag. The C code is output (using the appropriate substitutions) in lines 31–39. Line 32 substitutes the `$$` variable with its C equivalent, while the loop in lines 33–36 substitutes all `$1`–`$n` variables. The result of all substitutions is printed at line 37. Finally, line 40 generates the result return code.

7 Lessons Learned

In the following sections we give some specific hints on tool implementation and a critique of the system implementation approach by linguistic transformation tool building.

7.1 Hints on Tool Implementation

We found that the Perl language provided a suitable vehicle for implementing the tools. Its interpretative mode of execution meant that the tools could be built by rapid prototyping methods. The regular expression and associative array capabilities proved very useful when dealing with the program source text. Although Perl is a large, monolithic language rather than a small specific tool, its depth and breadth of coverage of many domains made development in it productive and fun. The execution speed of the Perl scripts was sufficient for all the tools.

Although parsing the free format language source seems the most natural approach for implementing a language processor we found that this can often be avoided. This makes it possible to develop tools whose existence could not have been justified given the time needed for developing a full language processor. Three methods can be used to get around the parsing overhead:

Use of lexical hints: The little languages developed were line-based with specific tokens marking the beginning and ending of the various blocks. In addition strict formatting and naming rules were imposed on the C language source in the cases where it would be processed by another tool (tree manipulation generators and error message management.)

Produce output that another tool will parse: The compiler for Haskell primitives did not parse the C code. It passed it directly to the C compiler for parsing. Suitable "`# line`" directives were put in the file so that syntax errors in the C source code would be reported with reference to the original file and line number rather than the automatically generated code.

Use an existing tool for parsing: The character class type compiler, used the regular expression capability of Perl to avoid parsing and verifying the regular expressions used. Another tool — not described here — in order to avoid parsing C code to search for global identifiers, compiled the C source into object code and examined the object code name list.

The burden of generating code for a custom language can be avoided by compiling into an existing language. C is suited as a target language. Its array initialisation facilities are suited for automatic code generation. In addition, as numerous tools have adopted this approach (*yacc*, *lex*, *twig*, C++ front etc.) the existing C compilers are less probable to complain when faced with machine generated output which is the sort of code that will impose the greatest stress on them. If the semantic gap between the little language and an imperative language is too wide to be bridged by a single tool, the tool can output code for another tool such as *lex*, *yacc*, LaTeX, or even a custom-made one.

7.2 Tool Building Advantages

We found the following advantages in using the specialised tool building approach for implementing parts of the Haskell system:

- The specification of a system part is isolated and easy to express.
- The linguistic gap between specification and implementation is narrowed as the specification is often the implementation (modulo the tool that transforms it.)
- Changes are easy to make, as implementation detail is not visible.
- Code becomes easily parameterisable. For example a number of different machine description files could be used to produce back-ends for different machine architectures.
- Code generated by this approach is often more efficient than other parameterisable approaches as it is compiled and not interpreted at runtime.
- Optimisations are easier to implement as they can be put into the little language translator and affect all the code written in that language.
- Repetitive use of similar constructs is avoided.

7.3 Tool Building Disadvantages

Working with custom-made tools made us realise a number of hidden costs and disadvantages:

- One additional level where things can go wrong is created. One may need to debug the translator as well as the little language code.
- The little language can become a source of confusion for people that come across it and are not used in this mode of development.
- Porting difficulties can be experienced as one needs to port the translators as well as the system. This means that the languages in which the translators are written must exist on the target system. This problem can sometimes be overcome by running the translators on another system and compiling the resulting code.
- Using a symbolic debugger on the automatically generated code can be difficult and unproductive.

In summary, we believe that building specialised tools in a language implementation effort, is a viable approach that can often result in enhancing the reliability of the system, while decreasing the development cost. In our case we found that this tool building philosophy, made the Haskell project realisation possible.

Acknowledgements

I would like to thank Sophia Drossopoulou, Susan Eisenbach, Paul Kelly, and the anonymous reviewers for their helpful comments on an earlier draft of this paper.

Support from the British Science and Engineering Research Council is gratefully acknowledged.

References

[1] Aho AV, Kernighan BW, Weinberger PJ (1988) *The AWK Programming Language*. Addison-Wesley.

[2] Bentley JL (1986) Little languages. *Commun. ACM*, 29: 711–721.

[3] Bentley JL (1988) *More Programming Pearls: Confessions of a Coder*. Addison-Wesley.

[4] Boehm HJ (1988) Garbage collection in an uncooperative environment. *Software: Practice & Experience*, 18: 807–820.

[5] Brooks FP (1975) *The Mythical Man Month*. Addison-Wesley.

[6] Brown JR, Cunningham S (1989) *Programming the User Inteface*. John Wiley & Sons.

[7] Brown PJ (1982) 'my system gives excellent error messages' — or does it? *Software: Practice & Experience*, 12: 91–94.

[8] Burstall R (1984) Programming with modules as typed functional programming. In *Fifth Generation Computer Systems 1984*, pp 103–112, Tokyo, Japan. Institute for New Generation Computer Technology (ICOT), North-Holland.

[9] Douglas RT (1990) Error message management. *Dr. Dobb's Journal*, pp 48–51.

[10] Ducassé M, Emde AM (1991) Opium: A debugging environment for Prolog development and debugging research. *ACM Software Engineering Notes (SIGSOFT)*, 16: 54–59. Demonstration presented at the Fourth Symposium on Software Development Environments.

[11] Field AJ, Harrison PG (1988) *Functional Programming*. Addison-Wesley.

[12] Glaser H, Hartel P, Wild J (1990) A pragmatic approach to the analysis and compilation of lazy functional languages. Technical report CSTR 90-10, Department of Electronics and Computer Science, University of Southampton. (To appear in Proc. of the Workshop on Parallel and Distributed Processing, Sofia, 1990. North-Holland 1991).

[13] Grosch J, Emmelmann H (1991) A tool box for compiler construction. In Hammer D, editor, *Compiler Compilers : Third International Workshop, CC '90*, pp 106–116, Schwerin, Germany. Springer-Verlag. Lecture Notes in Computer Science 477.

[14] Grosh J (1989) Ast — a generator for abstract syntax trees. Compiler Generation Report 15, GMD Forshungsstelle an der Universität Karlsruhe, Germany. (Revised Version).

[15] Hadjicocolis A (1990) Generating G-machine code from Haskell. Project report, Imperial College, Department of Computing, London, UK.

[16] Hudak P (1989) Conception, evolution and application of functional programming languages. *ACM Comput. Surv.*, 21: 359–411.

[17] Hudak P, Jones SP, Wadler P, Boutel B, Fairbairn J, Fasel J, Guzmán MM, Hammond K, Hughes J, Johnson T, Kieburtz D, Nikhil R, Partain W, Peterson J (1992) Report on the programming language Haskell, a non-strict, purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27. Haskell Special Issue.

[18] Johnson SC (1975) Yacc — yet another compiler-compiler. Computer Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, USA.

[19] Johnson SC, Lesk ME (1987) Language development tools. *Bell System Technical Journal*, 56: 2155–2176.

[20] Kernighan BW, Pike R (1984) *The UNIX Programming Environment*. Prentice-Hall.

[21] Kernighan BW, Plauger PJ (1976) *Software Tools*. Addison-Wesley.

[22] Kernighan BW, Ritchie DM (1982) The M4 macro processor. In *UNIX Programmer's manual: Supplementary Documents*, volume 2, pp 433–439. Holt, Rinehart and Winston, seventh edition.

[23] Lamport L (1985) *LATEX: A Document Preparation System*. Addisson-Wesley.

[24] Lehman MM (1991) Software engineering, the software process and their support. *Software Engineering Journal*, 6: 243–258.

[25] Lesk ME (1975) Lex — a lexical analyzer generator. Computer Science Technical Report 39, Bell Laboratories, Murray Hill, NJ, USA.

[26] Paxson V (1989) *Flex: Fast Lexical Analyzer Generator*. Real Time Systems, Bldg, 46A, Lawrence Berkeley Laboratory, Berkeley CA, USA.

[27] Peyton Jones SL (1987) *The Implementation of Functional Programming Languages*. Prentice-Hall.

[28] Schmidt DC (1990) Gperf: A perfect hash function generator. In *USENIX C++ Conference*, pp 87–100, San Francisco, CA, USA. Usenix Association.

[29] Spinellis D (1990) An implementation of the Haskell language. Project report, Imperial College, Department of Computing, London, UK.

[30] Stallman RM (1992) Using and porting GNU CC. Free Software Foundation, 675 Mass Ave, Cambridge, MA, USA.

[31] Suzuki N (1982) Analysis of pointer “rotation”. *Commun. ACM*, 25: 330–335.

[32] Tanenbaum AS, Kaashoek MF, Langendoen KG, Jacobs CJH (1989) The design of very fast portable compilers. *ACM SIGPLAN Notices*, 24: 125–131.

[33] Tjiang SWK (1986) Twig reference manual. Computer Science Technical Report 120, AT&T Bell Laboratories, Murray Hill, New Jersey, USA.

[34] Tsahageas PA (1990) Type checking for Haskell. Project report, Imperial College, Department of Computing, London, UK.

[35] Turner DA (1979) A new implementation technique for applicative languages. *Software: Practice & Experience*, 9: 31–49.

[36] Wall L, Schwartz RL (1990) *Programming Perl*. O’Reilly and Associates, Sebastopol, CA, USA.

[37] Warren DHD (1983) An abstract Prolog instruction set. Technical Note 309, SRI International, Artificial Intelligence Center, Computer Science and Technology Division, 333 Ravenswood Ave., Menlo Park, CA, USA.

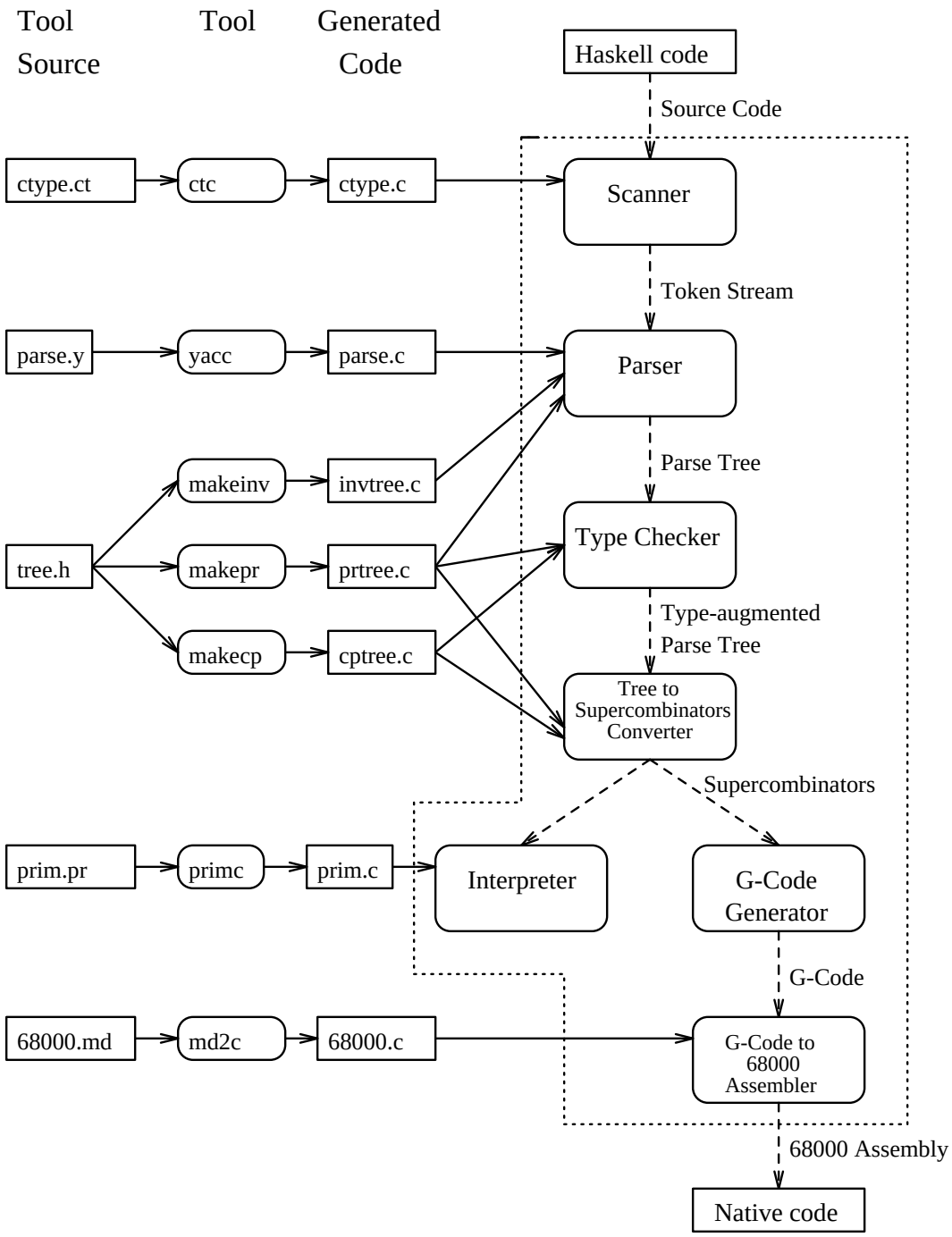

Figure 1: System Construction, Structure and Cooperation of the Components

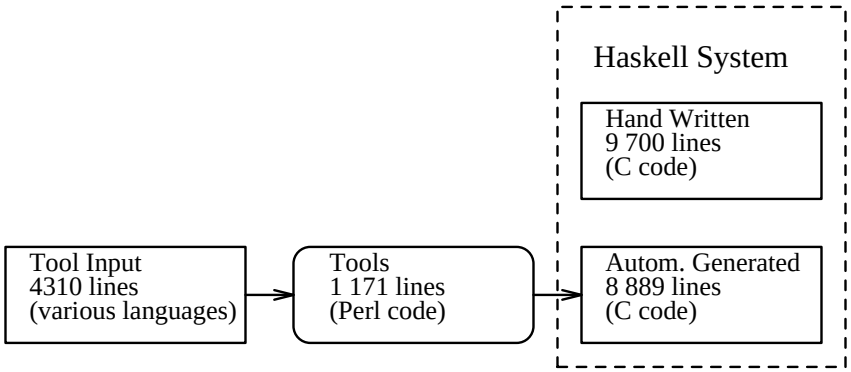

Figure 2: Division of System Code

ctype: `ct_Y`= 2^5 `ct_H`= 2^1

	31		5		1	0
0						
3			0		1	
4			0		1	
@			1		0	
A			0		1	
B			0		1	
255						

Figure 3: Character classes as a packed bit-vector table

```
 1 open(IN, "<prim.pr"); open(OUTC, ">cprim.c");
 2 while (<IN>) {
 3     if (/^%type\s/) {
 4         s/\s+$//g;
 5         ($dummy, $haskell, $c, $enum, $union) = split(/\s*:\s*/);
 6         $ctype{$haskell} = $c;
 7         $cenum{$haskell} = $enum;
 8         $cunion{$haskell} = $union;
 9     } elsif (/^primitive\s/) {
10         s/^primitive\s*//;
11         s/\s+//g;
12         ($name, $htypestr) = split(/::/);
13         @htypes = split(/->/, $htypestr);
14         push(@prims,$name);
15         $arity{$name} = $#htypes;
16         print OUTC "
17 static struct s_expression *
18 $name(struct s_expresslist *ppargs)
19 {
20     struct s_expression *ppresult = xmalloc(sizeof(*ppresult));
21 ";
22         for ($i = 0; $i < $#htypes; $i++) {
23             print OUTC "struct s_expression *pparg$i;\n";
24         }
25         for ($i = 0; $i < $#htypes; $i++) {
26             printf OUTC "pparg$i = eval(ppargs->%sexpr, NULL);\n",
27                         'next->' x $i,
28                         "assert(pparg$i->kind == $cenum{$htypes[$i]});\n";
29         }
30         print OUTC "ppresult->kind = $cenum{$htypes[$#htypes]};\n";
31         while (<IN>) {
32             s/\$\$/ppresult->u.$cunion{$htypes[$#htypes]}/g;
33             for ($i = 0; $i < $#htypes; $i++) {
34                 $varname = '\$' . ($i + 1);
35                 s/$varname/pparg$i->u.$cunion{$htypes[$i]}/g;
36             }
37             print OUTC;
38             last if (/^\}/);
39         }
40         print OUTC "return ppresult;\n}\n\n";
41     }
42 }
```

Figure 4: Compiler for Haskell Primitives