CHIRP: A Front End for Charity

Marc A. Schroeder

Department of Computer Science
University of Calgary
CANADA

email: marc@cpsc.ucalgary.ca

April 2, 1993

Abstract

Chirp is the front end of the interpreter for **Charity**, a *categorical* programming language currently under development. Chirp interfaces neatly with Barry Yee's **Charm** abstract machine, the back end, to form a coherent system for the definition and execution of Charity programs. Chirp has been designed to be *fast*, *portable*, and *extensible*.

Chapter 1 provides a brief introduction to Chirp and the Charity language. Chapter 2 presents Charity in slightly more detail, discussing categorical programming as well as issues in the language. Chapters 3–9 explore the workings of the Chirp system in detail, while chapter 10 explains the Chirp/Charm interface. Finally, chapter 11 comments on some possible directions in the continued development of Chirp and Charity. The appendices specify the grammar for Charity, and other technical or useful details of the system and language. An example Chirp session is also listed.

0.1 Acknowledgements

I would like to thank Dr. Robin Cockett for his support, enthusiasm, and many insights into the development of Chirp. Special thanks is owed to Tom Fukushima, who spent countless hours tutoring me in several details of the theory behind Charity, none of which he was obligated to spend.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 What is CHARITY?

Charity[4] is a new programming language, invented by Dr. Robin Cockett, currently under development at the University of Calgary. It is what is said to be a *categorical* programming language, although its "look and feel" is that of a *functional* language. The notion of a categorical language (a language based upon *category theory*) is explored briefly in chapter 2. However, the reader likely has greater familiarity with the notion of a functional language, so Charity's relation to that programming paradigm is examined here.

Like most functional languages, the basic programming methodology in Charity is simple: The user thinks in terms of defining *datatypes*, and then defining *functions over* those datatypes. One may then *compose* these functions to build new functions (and thus *programs*). Execution of a program is not thought of as the passing of control to a list of machine instructions, but rather as the evaluation of an *expression*. Besides the input initially provided to a function when invoked, and the result it produces, it has no other interaction with the "outside world". That is, there are no *side effects* in Charity (eg. no global variables, functions may not remember their "state", there is no

interaction with a *store*, there is no side-effected I/O, etc.).

In functional languages, *recursion* over recursively-defined datatypes is central to computations in those languages. Charity is not much different, although "recursion" in Charity is implemented in a much more controlled fashion. The reader is directed to chapter 2 for a more fully detailed explanation.

Charity has the following features common to functional languages:

Pure Functionality Functions may have no side effects or state. This means there are no global or static variables. The programmer is less concerned with the ideas of flow of control, storage space for data, referencing or dereferencing of data, etc., as she is in laying out an algorithm itself. In other words Charity is *declarative* as opposed to *imperative*.

Datatypes The programmer defines *arithmetic datatypes*, possible recursive in nature, in terms of *constructor* or *destructor* functions. Any needed type may be so defined, even infinite datatypes.

Strong Typing The typing of a function is *statically* inferred by the Charity system when defined. Any subsequent data passed to, or returned from, the function must *unify* with that *type signature*.

Polymorphism Functions may be defined such that the types of their input and output are *polymorphic* (ie. these types can vary under appropriate circumstances, but still must unify with the function's general type signature).

Pattern Matching This feature is still being developed, but hopefully functions will soon be able to decide which case to evaluate based upon the form of the input—This is already possible, but in an expressively more cumbersome way, with repeated use of the *case* operation.

Macros (Similar to "higher order functions")—Functions may take other functions (or *abstractions*) as parameters, to be used as *macros* within that function, with some limitations.

Laziness During evaluation of expressions, Charity attempts to take computational "shortcuts" when possible, by only computing what it absolutely needs to, and by sharing data. The reader is referred to chapter 10 , and [23].

It is hoped that the advantages of using Charity will become evident. Besides the usual advantages associated with declarative languages (such as rapid prototyping, concise and elegant representation of algorithms, ease of debugging, simplicity of language constructs and the underlying programming philosophy, and greater ease of constructing proofs), the following additional benefits, resulting from Charity being a categorical language, are noted:

- Charity is *strongly normalizing*. That is, programs written in the language *always* terminate, yielding some output. Thus, certain sources of subtle incorrectness are not present.
- Because Charity is based solely on category theory, there is a direct marriage between discrete mathematics and code. One may move easily from one to the other, hopefully (ultimately) allowing for increased ease of specification and verification.
- From the above points, then, we can see that proofs about Charity programs in general (ie. correctness, equivalence, etc.) are easier due the direct correspondence between mathematics and code, as well as no longer needing to prove that programs terminate.

For a more elaborate description of the Charity language, readers are directed to [4].

It should be noted that Charity is, again like most functional languages, an interpreted language. This point brings us to the next issue.

1.2 What is CHIRP? An Overview:

Chirp[1] (*CH*arity *I*nte*RP*reter) is, as the name suggests, an interpreter for the Charity language. More precisely, it is the *front end* of a larger system for the definition and execution of Charity programs. That is, Chirp interfaces to the **Charm** *back end* (*CH*arity *A*bstract *R*eduction *M*achine, Charm has been developed alongside Chirp by Barry Yee).

Chirp is what the user, or programmer, interacts with directly. Chirp allows the user to define datatypes and functions, evaluate expressions, and the like. Chirp also provides a set of commands so the user may manipulate the system itself.

Although Chirp is an interpreter, it is also in essence a compiler. The basic structure of the system is outlined here. Chapters 3–9 will describe the workings of Chirp's modules in greater detail.

Definitions and expressions (and in fact, any input from the user) is first fed through a *lexical analyzer*, or "scanner". The purpose of the scanner is to provide a stream of tokens to the *parser*, which analyzes the syntactic structure of the input. The parser in turn directs the building of intermediate structures for the representation of the input. Simultaneously, the parser directs the addition of information about identifiers to the *symbol table*. Thus, when parsing of some definition or expression is complete, we have not only inspected its syntactic structure, but we have also recorded various details regarding the identifiers in the input, and built tree-structures which represent the input. Any errors encountered in the input, except typing errors, will also have been caught and reported by the end of parsing. Chirp then applies a translation algorithm to the intermediate representation, to compile it into a stream of *categorical combinators*. These are linear, variable free machine instructions for Charm. When evaluating an expression, the combinators corresponding to the expression are passed to Charm, which then rewrites them into normal form (ie. some canonical form which represents the result of the evaluation), with a guarantee that the rewriting process will terminate.

[1] See appendix G for references to source and an executable system.

These resulting combinators are translated back into the high-level *term logic* which the user programs in, to be displayed as the answer. When making function definitions, the corresponding combinators are simply stored in expectation of later use.

1.3 Why CHIRP?

A prototype Charity interpreter[2] had already existed prior the the commencement of the development of Chirp. This fine piece of work was written in SML by Tom Fukushima, under the direction of Robin Cockett [12]. As an experimental system, it fulfills its obligation well. However, the system is a little spartan, and lacks niceties which would be invaluable in a system which would see widespread use. Chirp is a logical progression in the development of the Charity language.

Chirp has all the functionality of it's predecessor. Additionally, it provides a comfortable, consistent, and powerful environment for the development of Charity software. Attention has been paid to improving the manner in which the user interacts with the system. Some of these are:

- The ability to invoke an editor from within the system.
- Querying of the user's environment under Unix, so that it may be conformed to.
- Increased ease of cutting and pasting under windowing software.
- Builtin "help" and "info" screens.
- The ability to make definitions "on the fly" from within the interpreter, as well as from Charity source files (a trend which has not been followed in other modern functional languages, but which is believed useful).

Some of the "look and feel" of Chirp has been inspired by the interpreter for the functional language Miranda.

[2]The old prototype version of the Charity system, along with example programs and relevant papers, is available via anonymous ftp from `fsa.cpsc.ucalgary.ca` in `pub/charity`.

Although many of its additional features seems small, it is believed that overall, they contribute to Chirp's status as a user-friendly system.

The current version of Chirp has been developed in Ansi C, under the Unix operating system. This, along with the design decisions outlined in the rest of the paper, result in Chirp being:

Fast Although most of the speed issues in the interpreter rest on the abstract machine (which is beyond the scope of Chirp itself, eg. lazy evaluation), it is still the case that Chirp does not possess the slow, cpu-intensive behavior which is noticeable to the user in the prototype.

Portable A main advantage of the development of Chirp is that it will now be possible to port Charity easily to a variety of other platforms. Hopefully this will increase the use of the language at other sites, and make the system more popular.

Extensible It is expected that development on Chirp will continue, and thus it has been designed such that new features in the language can be built around the current working system.

Chapter 2

Preliminaries

With programming languages named "Hope" and "Faith" already in existence, developing a new language called "Charity" perhaps isn't too strange[1].

This chapter seeks to explain some of the more language-theoretic aspects of Charity itself. A detailed coverage of any of the material mentioned here is far beyond the scope of this paper. The aim is, however, to introduce some key concepts, and to refer the reader to more detailed treatments of the subject matter.

We begin by introducing a branch of discrete mathematics called *category theory*, and then illustrate the use it has been put to in computer science (especially the use of category theory, directly, as a programming language). We then outline some aspects of Charity itself.

2.1 Categorical Programming Languages

Definition. A **category** C consists of:

- A set of *objects*— $obj(C)$

[1] As for the name of the interpreter, isn't "chirping" what robins do?

- A set of *morphisms* (or *arrows*)— $arr(C)$

where:

1. Each morphism has exactly one *domain* and *codomain* from $obj(C)$.

   When the domain of morphism f is X and the codomain is Y, we write:

   $$f : X \to Y$$

2. $\forall X \in obj(C)$, there is exactly one *identity* morphism:

   $$1_X : X \to X$$

3. Given morphisms $f : X \to Y$ and $g : Y \to Z$, there is exactly one *composite* morphism:

   $$g \circ f : X \to Z$$

4. The following must be satisfied:

   - *Identity Laws*. If $f : X \to Y$ then

     $$1_Y \circ f = f = f \circ 1_X$$

   - *Associative Laws*. If $f : W \to X$, $g : X \to Y$, and $h : Y \to Z$ then

     $$h \circ (g \circ f) = (h \circ g) \circ f : W \to Z$$

Category theory, then, is the study of the properties of categories. This is a rich field in its own right, but lately has been applied in the domain of computer science quite heavily [20]. One such application is the use of category theory to argue about the semantics of a programming language. Another is the use of categories

directly as programming languages. Languages of this kind are known as **categorical programming languages**, and were first proposed by Hagino[14].

To see how this works, consider that one could model computer software with a category. *Datatypes* can be represented by *objects*, while *functions* (or programs) can be represented by *morphisms*. Intuitively:

1. Functions map one datatype to another.
2. One may always construct the identity function which maps a datatype to itself.
3. One can compose functions to build up programs.
4. Composing a function with the identity is the same as applying that function by itself, and the way we associate the composition of functions in a program does not affect the resulting program we have constructed.

Many of the properties of functional languages now work out quite nicely in this model. Briefly:

- The model is inherently purely functional.
- Arithmetic datatypes can be defined in terms of constructors [destructors] simply in terms of some morphisms into [out of] the object for that new type we are defining.
- Strong typing is necessitated. This means that when we define a new function out of a composition of "smaller" functions, we must always initially check that the domains and codomains of these functions "line up". Otherwise, the composition can't really be accomplished at all.
- Using *functors* (a mapping over categories) we arrive at the ability for having polymorphic datatypes.
- Important features in categories, such as *products*, *coproducts*, the *dual* of a category, etc., have natural interpretations in programming languages. For instance,

products will correspond to pairs, and through them we can build up records. Dual to this idea is that coproducts can be used to create variant records.

If we are to use category theory for the statement of programs and expressions, we also need some kind of method for evaluating expressions. We may specify an *abstract machine*, similar to the machines used to evaluate other modern functional languages, to this end. Such a machine will *rewrite* a given expression (which is just the composition of morphisms) into its *normal form*—this is taken to be the result of the evaluation. Note that the machine itself is "categorical", in that we derive its set of *rewrite rules* from the properties of categories. These functions that we have composed, and which we provide as "machine code" for the abstract machine, are called **categorical combinators** (as they are variable-free). An operational semantics for a Charity abstract machine is given in [4].

The reader will recall that for traditional functional languages, evaluation of expressions is equivalent to normalization in the lambda calculus, and that this process of "reduction" is not always guaranteed to terminate. However, there is no interpretation for *fixed point operators* in our rewrite rules when they are based on category theory. Since it is the existence of fixed point operators in the lambda calculus which raises the possibility of nontermination, categorical programs are *always* guaranteed to terminate! We say that categorical programming languages are **strongly normalizing**.

A form of recursion will exist in categorical programming languages, but it will be recursion which will *always* be guaranteed to "bottom out".

The above points mean that categorical programming languages cannot compute all computable functions, but as long as we can place an upper bound on the computation time to solve a problem, it is doable. The tradeoff is, however, that we may construct elegant categorical languages which have the nice theoretical properties that:

- A subtle source of error, nontermination, has been eliminated. Proofs of termination are never required when constructing proofs about a program, or working with the semantics of the language.

- Because of the guarantee of termination, we may apply the proof techniques of *structural induction* and *case analysis* for programs in the language. This is complicated in other languages because of the existence of *fixed points*.

- There is a much more direct correspondence between the mathematical basis of the language and programs in the language themselves. Again, this simplifies proof-making, and enhances the ability to specify and verify problems and solutions.

- Again, the functional language paradigm can be elegantly modeled by categories, and thus categorical programming languages can have all the theoretical benefits the computer scientist has come to expect of them.

The above does not mean that we cannot have infinite datatypes in categorical languages. In fact, a very complete theory of **strong categorical datatypes** has been developed (for use in Charity) [8][9] which allows us to define any kind of datatype we wish—finite (**inductive**) datatypes built up using *constructors*, or (potentially) infinite (**coinductive**) datatypes taken part using *destructors*. What the above actually means is that computations over infinite datatypes are still guaranteed to terminate.

2.2 Charitable Notions

Charity is a categorical programming language on the leading edge of the development of the theory[2]. It is the belief of Marc Schroeder that "Charity" is an even more appropriate name for the language when one considers that it provides the computer scientist with all the niceties given section 2.1.

At first, Charity is assumed to have only the most primitive types and combinators available (see appendix C for a list). The user explicitly defines exactly what is needed.

Inductive datatypes are given by definitions of the form:

[2] $IMP(G)$, a project of Bob Walters in Sydney, is another known example (however, it uses an imperative paradigm, and is probably currently less well developed).

$$\begin{array}{ll} \textbf{data} & L(A) \longrightarrow S = \\ & c_1 : E_1(A, S) \longrightarrow S \\ & \vdots \\ \mid & c_n : E_n(A, S) \longrightarrow S. \end{array}$$

Where L is the name of the type, A is a finite product of other arbitrary datatypes on which the L is based, and each c_i is a constructor for the datatype.

Coinductive datatypes are given by definitions of the form:

$$\begin{array}{ll} \textbf{data} & S \longrightarrow R(A) = \\ & d_1 : S \longrightarrow F_1(A, S) \\ & \vdots \\ \mid & d_n : S \longrightarrow F_n(A, S). \end{array}$$

Where R is the name of the type, A is as for inductive definitions, and each d_i is a destructor for the datatype.

Here are some example datatype definitions as would be entered by a programmer:

```
% inductive def'n for booleans:

data bool -> B =
      true  : 1 -> N
    | false : 1 -> N.

% inductive def'n of natural numbers:

data nat -> N =
      zero : 1 -> N
    | succ : N -> N.

% inductive def'n for lists:

data list(A) -> L =
      nil  : 1     -> L
    | cons : A * L -> L.

% coinductive def'n for infinite lists:
```

```
data I -> inflist(A) =
      head : I -> A
    | tail : I -> I.
```

For each inductive datatype definition we can perform a *case* operation, a *fold* operation, and a *map* operation (when an inductive datatype is defined, functions `case_type`, `fold_type`, and `map_type` are actually added to the system). Note that all three could be expressed in terms of a fold.

The **case expression** is written as:

$$\text{case}^L\{f_1, ..., f_n\} : L(A) \times X \longrightarrow Y; (z, x) \mapsto \left\{ \begin{array}{ccc} c_1(z_1) & \mapsto & f_1(z_1, x) \\ & \vdots & \\ c_n(z_n) & \mapsto & f_n(z_n, x) \end{array} \right\} (z)$$

where $f_i : E_i(A, L(A)) \longrightarrow Y$.

The case expression allows the programmer to express conditionals. The programmer could enter something like:

```
def is_empty(l) =
    { nil  => true
    | cons => false
    }
      (l).
```

The **fold expression** is written as

$$\text{fold}^L\{g_1, ..., g_n\} : L(A) \times X \longrightarrow C; (z, x) \mapsto \left\{ \begin{array}{ccc} c_1 : z_1 & \mapsto & g_1(z_1, x) \\ & \vdots & \\ c_n : z_n & \mapsto & g_n(z_n, x) \end{array} \right\} (z)$$

where $g_i : E_i(A, C) \times X \longrightarrow C$.

The fold expression allows the programmer to perform induction (bounded recursion) over data. The programmer could enter something like:

```
def add(x, y) =
    {| zero : ()  => x
     | succ : sum => succ(sum)
     |}
        (y).
```

The **map expression** will be written as

$$\text{map}^G\{h_1, ..., h_m\} : G(A_1, ..., A_m) \times X \longrightarrow G(B_1, ..., B_m); (z, x) \longrightarrow G \left\{ \begin{array}{ccc} y_1 & \mapsto & h_1(y_1, x) \\ & \vdots & \\ y_m & \mapsto & h_m(y_m, x) \end{array} \right\} (z)$$

where $h_i : A_i \times X \longrightarrow B_i$.

The map expression allows the programmer to transform data into some data of a similar form (eg. the example below will cause each `nat` in a list of natural numbers to be incremented by one—we are still left with a list). The programmer could enter something like:

```
def inc_list(l) = list{el => succ(el)}(l).
```

For each coinductive datatype definition we can perform a *record* operation, an *unfold* operation, and a *map* operation (when a coinductive datatype is defined, functions `record_type`, `unfold_type`, and `map_type` are actually added to the system). Note that all three could be expressed in terms of an unfold.

$$\text{record}^R\{f_1, ..., f_n\} : X \longrightarrow R(A); (x) \mapsto \left(\begin{array}{c} d_1 : f_1(x) \\ \vdots \\ d_m : f_n(x) \end{array} \right)$$

where $f_i : X \longrightarrow F_i(A, R(A))$.

The record expression allows the programmer to add new data onto some coinductive structure. The programmer could enter something like:

```
def fibonacci =
     (head: zero, tail: (head: succ(zero), tail: fib')).
```

The **unfold expression** is written as

$$\text{unfold}^R\{g_1,...,g_n\} : C \times X \longrightarrow R(A); (z,x) \mapsto \left(\left| z \mapsto \begin{array}{ll} d_1: & g_1(z,x) \\ \vdots & \\ d_n: & g_n(z,x) \end{array} \right| \right)(z)$$

where $g_i : C \times X \longrightarrow F_i(A, C)$.

The unfold expression allows the user to perform coinduction over infinite datatypes. The programmer could enter something like:

```
def fib' =
     (| (x, y) =>
              head : add(x, y)
      |       tail : (y, add(x, y))
      |)
         (zero, succ(zero)).
```

The **map expression** is the same as for the initial datatypes.

The above examples also provide examples of how one defines composite functions in Charity, with the `def` syntax.

The attentive reader will note that we are not explicitly linearly coding the functions as a string of compositions to be rewritten by an abstract machine. Rather we are making use of a high level syntax, called the **term logic**, which can be mapped using a well understood translation function to the categorical combinators (see chapter 8). This is much better for the programmer as it is structured, it allows the programmer to use variable names, and it frees him/her from the more category-theoretical details.

Besides the expressions (or **terms**) listed above, the user may also code the following terms:

- The empty term: ().

- The identity term: x (ie. the result of some expression is just a variable).
- A pair of terms: (t_1, t_2).
- Composition of terms: $f(t)$.
- Abstractions: $\{w \mapsto t_1\}(t)$ (this is a form of pattern matching, by which the result of term t matches the variables in w, such that t_1 can use those variables separately. This is useful for breaking up pairs).

Variable bases (such as a list of formal parameters in a function definition, or w in the term for abstractions above) bind free variables in terms, so that all variables in a valid program are defined (see section 8.2).

A feature which Charity has incorporated is that of *macros*. That is, functions may take abstractions as a special kind of parameter, to be applied within the function itself. However, macros are *not* first-class entities. That is, they may not be passed to other functions as macros in turn. Further, a function may not pass itself as a macro[3].

This brings us to a point regarding recursive definitions: Functions are not considered available for use in expressions until after they have been completely defined. That is, a function may not be defined in terms of itself, nor may a function refer to another which appears later in the input source text (recursive definitions must make use of the fold expression). This ensures that unbounded recursion and mutual recursion are not possible. Macros are not allowed to be first-class entities because if they were, it would be possbile for a function to pass itself to itself as a macro, violating our conditions.

Lastly we will note that *primitive recursive* functions are expressible in Charity. In fact, since Charity can compute Ackermann's function, it is more than primitive recursive. Still, it is limited by the constraint that we cannot evaluate functions which have a possibility of not terminating.

One can see that Charity is actually a very simple language, based on elegant principles, while being expressively powerful (up to possibly nonterminating functions).

[3]To illustrate this distinction, Charity syntax forces macro parameters to be passed to functions in a different list than other actual parameters.

Chapter 3

The Waiter

Chirp is an *interpreter*, and therefore must deal with the user on an interactive basis. The module which performs the task of interaction is called the **waiter**, to borrow a term from Scheme[1].

The waiter, although simple, is the main driver for the entire system. It consists of three parts:

- The *Construction* Section
- The *Waiter* Proper
- The *Destruction* Section

The various sections are described below.

3.1 Construction

The construction section is invoked when Chirp is started up, or upon reset. It initializes the system's internal data structures, optionally reads in a Charity source file to begin working with, and reports some information to the user.

[1] An example Charity session is provided in appendix E.

3.2 The Interactive Waiter

The waiter proper then repeatedly prompts the user, and invokes the parser to deal with the user's input. The user may either define some new datatype or function, or may enter an expression to be evaluated, or may issue some command to the interpreter itself. Once the parser has handled the user's input, it returns an **action** for the waiter to perform. In the case of a definition or expression evaluation, the action is null. Rather, actions correspond to commands which affect the system itself[2]. A full list of commands is presented in appendix D. A subset is listed and briefly explained here:

`dump` List all defined datatype and functions.

`edit` Invoke the default editor to manipulate source text.

`file` Read in a Charity source file.

`quit` End the Charity session.

`reset` Remove all user-defined datatypes and functions, and reconstruct the system.

query Get detailed information on a datatype or function.

The waiter ends this interactive cycling when the user issues the `quit` command (a session may also be terminated with `control-D`).

3.3 Destruction

When the user ends the current Charity session, the destruction section cleans up the interpreter's internal data structures, and exists.

[2] Although definitions "affect the system" in a sense, they have no associated action, since they are added to Chirp's internal tables *during* the parse, as opposed to after it has been completed and the parser has returned.

3.4 The Current Working File

One important feature of the waiter is that it is always dealing with a *current working file*. This is either specified on the command line when invoking Chirp, or can be specified/altered with the `file` command from inside the interpreter. If no working file has been given at start-up time, Chirp assumes the use of a default working file entitled "`scrat.ch`". Note that the current working file does not need to actually exist in the filesystem. However, should the user invoke the default editor to add source text to such a file, that file will be created with the given name. Otherwise, such a non-existent working file is equivalent to an empty existing working file.

Chapter 4

Lexical Analysis

The parser should not interact directly with the programmer's input source text. Rather, it should simply draw from a reliable stream of *tokens* which represent that input. The parser can then group the tokens into the appropriate syntactic structures, as is its function. The module of Chirp which performs this task is the **lexical analyzer**, also known as the "scanner".

4.1 The Tokens

There are unique tokens for each reserved word in Charity, as well as for each symbol (eg. `(` `)` `;` `*` ...). There is also a token for identifiers (*non*-reserved words), and for the special "end of file" condition. Since all user input is first filtered by the lexical analyzer, there is also a "command" token, which indicates that the user is issuing a command to Chirp. White space (ie. spaces, tabs, newlines, and comments) has no tokenized representation. See appendix A for complete details on the lexical structure of Charity.

The scanner does not scan the input all at once. Instead, scanning is directed by the parser (in fact, the parser directs every aspect of input source text translation). The interface between scanner and parser is a simple `Scan()` function which the parser

calls when it needs to examine the next token from the input. Tokens are never re-read. Once the parser has been passed a token, its next call to the scanner must return the next token in line. This is acceptable, as the parser is a *predictive parser* (see chapter 5 for more information—in fact, once any given character has been read from the input stream, it is never re-read).

4.2 Input Sources

The lexical analyzer can interact with the input source text in one of two possible ways:

Reading from a file: Parsing a file which contains Charity definitions and expressions, the scanner reads its input from that file (actually, a *memory-mapped* copy of that file).

Reading from standard-input: Since the user may make "adhoc" data or function definitions from the keyboard, while inside the interpreter, the scanner may also read from the standard-input.

The benefit of the duality in the nature of the scanner is obvious: It allows the user of Chirp to select either input from a file, or adhoc input "on the fly", while not affecting *how* that input is parsed and translated—The parser simply sees a stream of tokens to parse, regardless of the form of the input. This is viewed as a great convenience for Charity programmers, while interpreters for other languages (such as the Miranda interpreter) always force the user to edit a file to be used as input.

4.3 The Name Table

The lexical analysis module also fulfills a second important role. It contains Chirp's **name table**. This is a table with unique entries storing the name of each identifier which has been scanned to the current point in parsing. When the identifier token is scanned, it is given an associated data field which points into the name table, indicating

which identifier has been encountered. This allows "higher level" modules in Chirp to reason about identifiers without knowing their actual names. Rather, they can simply deal with the unique pointers to the relevant names.

4.4 Interface Routines

The lexical analyzer provides the following routines other Chirp modules:

`ConstructLex()` This is called by the waiter when initializing the system. It essentially sets up the name table.

`DestructLex()` This procedure does the opposite of `ConstructLex()` for cleaning up.

`PreLex()` This procedure is called before parsing input (for instance to let the scanner know the medium for input).

`PostLex()` This procedure cleans up the scanner after parsing.

`Scan()` This reads in the next token from the input stream.

`InsertName()` Other modules need to be able to add identifier names to the name table (ie. the symbol table when it is initializing itself).

`Line()` The scanner provides this routine so other modules may know which line of input it is reading (ie. for error reporting).

`ForcedQuit()` This routine allows the waiter to confirm when the user has typed `control-D`.

`IsAdhoc()` This routine lets other modules know what the input medium is.

4.5 Errors Trapped in the Scanner

The scanner tries not to generate errors, since most are more properly handled by a higher level module, and since it has no conception of the context it is in. However, a few errors must be reported from the lexical analyzer in order to ensure a reliable stream of tokens is being generated. These are an error message for when a file to read from cannot be located, and an error message for encountering the "end of file" condition while inside a comment.

Other oddities such as control characters in the input stream, an "end of file" appearning at the end of a line of text, and extra data after a terminating `(.)` symbol are treated as warnings and are ignored.

Chapter 5

Parsing

The behavior of Chirp's **parser** is central to it's operation. Besides fulfilling the traditional goal of a parser, which is to analyze the grammatical structure of the input source text and ensure that it agrees with the syntax of the programming language, it also performs the more general task of driving the entire interpreter.

The parser's tasks include the following:

- Supervising the error handler, and initiating appropriate semantic checks (in addition to the syntax verification it is also performing).
- Prompting the lexical analyzer to scan more source code, and return new tokens for the parser to process.
- Directing the building of the symbol table, which stores needed information regarding identifiers which have been read in.
- Directing the building of an intermediate data structure, which represents the logical structure of the input source text.
- Calling the translator at the appropriate time, which in turn compiles the intermediate representation into categorical combinators.

- Calling the typechecker, which *unifies* the categorical combinator code.
- Calling the abstract machine, so that it may update its own tables following new definitions, or so that it may evaluate expressions which have been entered.

The details of these operations are covered in chapters 3–9. This section will concentrate on the actual act of parsing.

It should be noted that although the parser *directs* these operations, they are actually encapsulated in other code modules. Communication between modules is still tightly controlled through a small set of well defined interface calls, and thus the parser is not simply a mass of arcane code. The interfaces between the parser and the modules it controls are described in subsections below. First, however, we will examine the way in which the parser performs its primary task of analyzing the grammatical structure of Charity programs.

5.1 Predictive Parsing

Let G be a context-free grammar. Also, given some rule R of G, let *FOLLOW(R)* denote the set of all terminal symbols of G which can appear immediately following a substring derived by R. Then G is said to be an *LL(1)* grammar if and only if whenever $A \to \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

1. For no terminal symbol a do both α and β derive strings beginning with a.
2. At most one of α and β can derive the empty string.
3. If β can derive the empty string, then α does not derive any string beginning with a terminal in FOLLOW(A).

Note that no ambiguous or left-recursive grammar may be *LL(1)*.

Often, grammars which are initially presented such as not to be *LL(1)* can be transformed into such a grammar by removing left-recursion, and left-factoring. This is not always possible, however.

Grammars which are *LL(1)* share the useful property that one can construct *predictive parsers* which recognize them. Predictive parsers are a special kind of *top-down*, *recursive-descent* parser which never needs to backtrack. They simply scan the input from left to right, requiring only one token of look-ahead at all times. Intuitively this is because if we ever have a choice between productions to apply while recursively descending, a simply query of the **look-ahead token** will indicate which one to take.

If the grammar for a language permits us to write a predictive parser to recognize programs in the language, we have a definite advantage. This is because one may code such an extremely efficient parser for that language by hand, relatively easily. One potential danger, however, is that we may have an elegant but non-*LL(1)* grammar for a language, but if we can alter it such that it *is* of the appropriate form, that form is no longer a "nice" grammar to work with.

5.2 Predictive Parsing for Charity

One of the beauties of Charity is that its grammar is quite short and simple. Unfortunately, left-factoring and eliminating left-recursion is not sufficient to render it *LL(1)*. The good news is, however, that there are only a small number of productions in the grammar which violate the needed properties. Also it is satisfying to note that the left-factored, left-recursionless grammar we obtain is still simple and elegant. Furthermore, by making use of some contextual information while applying these rules in the parser, we can eliminate the ambiguity and deterministically make the right choices. This turns out not to be hard, and in effect allows us to write a fast predictive parser (or "near-predictive parser" at least) for Charity after all. In fact, since the parser has this property, we can make the entire system little more than a tight one-pass compiler, by allowing the parser to direct the other aspects of input source text translation.

The grammar used in the construction of Chirp's parser can be found in appendix B.

The reason that the Charity grammar is not fully *LL(1)* is that there are some rules at which the next production to apply during the parsing cannot be fully determined by

a one-token lookahead. They are[1]:

1. The ambiguity between whether to parse an *inductive* or a *coinductive* datatype definition.

2. The ambiguity between whether to parse a *record* operation or a parentesized expression whos first symbol inside the parentheses is an identifier.

3. The ambiguity between whether to parse an *abstraction* or a *case* operation, in an expression.

4. When we begin parsing an expression, and immediately encounter an identifier, there is no grammatical indication of whether we are to treat this as a function to be applied, a variable to be referenced, or the start of a *map* operation.

5. The ambiguity between *special* and *general* commands.

Since the syntax for inductive and coinductive datatypes is quite similar, we can solve problem 1 by simply carrying along state information denoting the form of what has already been scanned. Thus, when parsing data definitions, the parser is temporarily "context sensitive".

The style of Charity programs is such that all identifiers must be declared before they are used. For instance, the definition for the function `foo` must be made before any other functions may utilize it. A corollary of this fact is that whenever we scan an identifier as part of an expression, it is one which we already know something about (otherwise an "undefined identifier" error is flagged). Thus, upon reaching such a token, we can lookup pertinent information regarding it in our symbol table, and thus determine it's *class* (The class of an identifier states whether it is a *function*, *type*, *variable*, *arbitrary type* as part of a datatype definition, or the *representation* for a type being defined in a datatype definition). This yields enough information about what we

[1] To offending productions are labeled in appendix B. A brief examination should convince the reader that these ambiguities cannot be resolved by transformations on the grammar alone.

have scanned to suggest a production to apply, and thus solve problems 2–4, and help in a our solution to 1.

"Problem" 5 is hardly a problem at all. The initial confusion lies in the fact that Chirp has two kinds of commands: *general* commands, which may be invoked from within a file *and* from the Chirp prompt, and *special* commands, which are only valid when entered at the prompt (for instance, it is nonsensical to issue a `reset` command while reading in a file). A quick check of the grammar in appendix B will indicate that when parsing adhoc input from the keyboard, it is ambiguous whether scanning a command token should denote a call to the special command production, or the general command production. There is really no problem, however, since the "adhoc" procedure will *always* call the special command production in this case. Then the actual command which was typed will be looked up. If the command is valid everywhere the general command production is used to deal with the input, otherwise the special command production deals with it.

5.3 Coding the Parser

As stated above, Chirp's parser is a recursive-descent parser. That is, we construct a procedure for each non-terminal on the left-hand side of a rule in the grammar in appendix B. The body of that procedure is determined exactly from what appears on the right hand side: A non-terminal corresponds to a recursive call to another procedure; a terminal corresponds to a call to the scanner, since we must "match" that token to the current look-ahead token, read in a new look-ahead, and continue on. If there is ever a choice to make between productions to apply, we use a conditional construct to handle each contingency, based on what the look-ahead is (again, this is guaranteed to work since the parser is essentially predictive, unless the rule is one of the few ambiguous ones, in which case we also use some contextual information—a query of the symbol table for instance).

Any other duties that the parser must oversee are therefore handled by placing calls

to the relevant modules inside these procedures (eg. building up the symbol table). These procedures may synthesize attributes (data) from the results of recursively called procedures, and may inherit needed attributes from their callers as parameters. They may then in turn pass results back (eg. the intermediate structure being built up).

This need to do more than just *parse*, but actually *direct* the rest of the compilation process, means that the procedures in the parser must have some way to interface with other Chirp modules. This leads us to the next section.

5.4 Interfacing with Other Modules

It is now known how the parser goes about grouping the tokens into their hierarchical positions within the Charity grammar. But as we go about the syntactic analysis, this also gives Chirp cues as to when to perform other needed operations. These other tasks are handled by modules which are described in detail elsewhere in this report. However, the parser's interface to them is outlined briefly below.

5.4.1 On The Waiter Taking Orders

When the waiter calls the parser, it optionally passes it a filename to parse. The parser then tells the lexical analyzer to read its input from that file.

More importantly to the waiter, the parser passes back **actions** for the waiter to perform. These invariably correspond to user commands which the waiter must attend to once the parser has safely existed.

For instance: `quit`, `reset`, `edit`, etc.

There is also a *null* action, for when the parser has no command to pass back to the waiter (eg. after a datatype definition).

5.4.2 Error Handling

As the parser carries out its processing, it may encounter either *syntax* errors (the parsing action *itself* has failed due to the fact that input source text does not conform

to the Charity grammar), or possibly *semantic errors* (the information being built up about the input is incomplete or inconsistent, again due to some problem in the input source text—eg. The typechecking of some function fails). When this happens, the parser needs some way of flagging an error, and possibly aborting the compilation.

Functions provided by the error handler:

`ReportError()` Allows the parser to flag erroneous input source text, giving the user some indication of what has gone wrong.

`ReportBadCommand()` Allows the parser to inform the user that an invalid command was entered.

Semantic Checks at Parse-Time

Input source text which passes syntactic checks must still contend with semantic checks. Most of this is done by the typechecker after the input has been read and translated (see chapter 9). One of the beauties of Charity is that those error checks can be systematically described (see figure 9.1), so that the tasks that an interpreter must perform are well defined.

There remain a small set of errors which must be checked for, as the parser continues its syntax directed translation. These are checks such as:

- Ensuring that identifiers which are encountered in expressions have already been defined, or that identifier names don't conflict.
- Making sure that fold, case, etc., expressions have been fully defined for all their constructors or destructors.
- Ensuring that when a function is being passed macros, the correct number of macros, according to its definition, are being passed.

Being able to assume these semantic checks have been carried out during parsing simplifies work later on (ie. The translation module assumes the structures which have

been built are correct, and so concentrates only on translating, and typechecking is easier as well).

5.4.3 Driving the Lexical Analyzer

The parser has only one main interface to the lexical analyzer: The `Scan()` function, which generates a new look-ahead token.

5.4.4 Building up the Symbol Table

One of the most important tasks the parser has is to build up information about identifiers in the input source text, and insert that information into the symbol table. The symbol table provides the following basic routines the that end:

`PushScope()` Push a new lexical scope for identifiers onto the symbol table stack.

`PopScope()` Remove a previously pushed lexical scope.

`LookUp()` Find, if possible, an instance of a given identifier at the highest lexical scope possible.

`Insert()` Insert a new identifier, and information about it, into the symbol table at the topmost lexical scope

5.4.5 Invoking the Translator

Another essential task of the parser is to supervise the building of an intermediate data structure. Once that structure has been built, the translator module translates it into categorical combinators. It provides the `Translate()` function so that the parser may invoke it at the proper time. It then passes the resulting combinator code back to the parser.

5.4.6 Calling the Typechecker

When the parser gets combinator code back from the translator, it calls the `TypeCheck()` routine provided by the typechecker. This is a semantic check, but also allows the system to infer and record the *type signature* for a new function.

5.4.7 Communicating with the Back End

After a function definition has been made, or an expression has been entered, and we have compiled that input into categorical combinators, typechecked it, and possibly updated the symbol table with new information, it is time to invoke the abstract machine. Charm provides the parser with two functions:

`Compile()` In the case of a function definition or an expression evaluation, we call this function so that Charm can update its own internal tables, in anticipation of a later request for an evaluation.

`Machine()` When actually evaluating an expression, we then call `Machine()` to carry out the computation.

Chapter 6

Error Handling

As the parser drives the syntax directed translation of the input source text, there exists the possibility of detecting an error. It is the task of Chirp's **error handling module** to report the error, and decide on a course of action.

6.1 Three Levels of Error

There are three kinds of error detectable in the input. They are:

warning Something strange has been encountered, but there is no concrete evidence to conclude that the input source text is definitely flawed. Action: Flag the warning, but otherwise continue.

error A definite error has been detected (either a syntax error—an unexpected token has been scanned, or a semantic error—for instance a new function definition fails to typecheck). Action: Flag the error and enter an "error state" in which we continue to parse the input file (to try to find more errors), but otherwise stop generating code. When the parse finally aborts, we also purge the symbol table of anything which was recently defined, to keep it from being corrupted.

fatal As an error, except that we abort the terminating process entirely. This indicates Chirp has assumed that the error cannot be recovered from. Note that after receiving some set number of normal errors, we treat the last error as fatal and abort.

6.2 Interface Routines

The interface routines provided by the error handler are simple. They are:

`PreError()` This routine is called before parsing commences, to initialize the error handler.

`PostError()` This routine is called after the parsing ends, to cleanup the error handler.

`Errors()` This routine returns a simple boolean value indicating if we are in the error state. This allows the translation module to know when to stop generating code, and allows the symbol table to know if it should purge itself of probational definitions at the end of a parse.

`ReportError()` This routine is called when a warning, error, or fatal needs to be flagged. It accepts the error kind, the line number of the error (which may be "undefined"), and a short, helpful message to be displayed for the user.

`ReportBadCommand()` This routine offers an alternative to `ReportError()`, when reporting an erroneous command. It is a simplified interface.

`Panic()` This routine is one which should never be called. Chirp is very careful about performing self checks on its own internal structures, as well as the results from the calls to operating system primitives and Charm. Chirp is especially vigilant of suspect data being passed between its modules via the various interface routines. `Panic()` is called when something strange happens internally to Chirp *itself*. When panicing, the system attempt to inform the user of the reason for aborting

(ie. whether Chirp is corrupt, or whether the operating system is failing), then tries to give some more diagnostic information (including a message as to what the specific incongruity was, and which line and Chirp source file caught the error). The system then aborts.

6.3 Error Recovery

Currently, Chirp considers almost all errors as fatal. This is partly justifyable, since it seems to be the trend in modern functional languages. This choice was also made in order to free up more development time for focusing on more important issues. It should be noted, however, that the error handler has been coded in a general fashion (ie. with full handling of errors and warnings—not just fatals), such that extending Chirp to have more sophisticated error handling later will not require a redesign of the module.

Chapter 7

Storage of Symbols and Important Data

As we parse input source text, we encounter **identifiers** [1]. These are essentially names to which we associate some kind of information. The reasons we need to build up and store such information are that:

- Such information may later be needed to direct parsing, and the further buildup of more data.
- Such information is needed in and of itself, and is thus the goal of parsing in the first place (for instance, the combinator code which defines the operation of some function—the result of translating some input source text).

The **symbol table** module is the data structure, along with related routines for manipulating that structure, which handles the storage and access to this information.

[1] See appendix A for the lexicographical characterization of an identifier token.

7.1 Needed Information

For each identifier we encounter, we will need to either create or lookup a corresponding **symbol table entry** for it. Therefore we must first characterize these entries.

Each entry is a record of information. All entries share the same following basic information:

name The name of the corresponding identifier. This is the key by which entries are looked up.

class This denotes which sort of identifier we are dealing with. There are 5 sorts: **type**, **function**, **variable**, **arbirary**, and **representer**.

level Each entry knows at which lexical scope it is defined.

from file An entry has a boolean flag which indicates whether it was defined in a file, or whether it was defined adhoc. This information is used when re-reading files, so that old definitions from the file may be first purged, while leaving adhoc definitions in place.

probational This is another boolean flag. Before Chirp is done parsing some input source text, all newly added entries are stored as being probational. If an error is detected, then when we exit the parser the symbol table is purged of all probational entries, since they must be assumed to be in error. Otherwise, probational entries revert to being non-probational.

7.1.1 The Classes of Identifiers

Entries also store other types of information, based upon the class. Below we describe what each of these classes mean, and what we need to know about instances of them.

Types

```
data list(A) -> L =
      nil  : 1      -> L
    | cons : A * L -> L.
```

Figure 7.1: An example datatype definition (for lists).

Given the example datatype definition in figure 7.1, the identifier `list` is of class *type*. In general entries of this class are created whenever new datatype definitions are made.

We need to record the following information for types:

- A flag indicating whether the type is *inductive* or *coinductive* (in this example, `list` is inductive).
- An integer telling us the number of *arbitrary types* the type refers to (in the example, `A` is our one (zeroth) arbitrary type).
- An integer telling us the number of *constructors* (or *destructors* for coinductive types) for the type (`list` has two constructors: `nil` and `cons`).
- An array of the names of each constructor (or destructor) for the type.

Functions

An entry of class *function* is added to the symbol table whenever some new function is defined. There are different ways in which definitions may occur, and figure 7.2 uses one method (the `def` syntax). Here, `member` will be added to the symbol table as a function.

The various kinds of functions, all defined differently, are described below:

primitive These are functions which are added into the system at construction time, at the lowest lexical scope. Examples are `id` and `pair` (a list is given in appendix C).

```
def member{eqFunc}(el, lst) =
    {| nil  : ()
           => false

     | cons : (el', found)
               =>
                  { true  => true
                | false => eqFunc(el, el')
                }
                  (found)
     |}
        (lst).
```

Figure 7.2: An example function definition (for determining membership in lists).

structor These functions are either *constructors* or *destructors* for a given type, and are defined during a datatype definitions. For example, `nil` and `cons` in figure 7.1.

ductive These are the *fold*, *case*, and *map* operations for a given *inductive* datatype, or the *unfold*, *record*, and *map* operations for a given *coinductive* datatype. When we define `list`, for example, in figure 7.1, the following maps are added to the system: `fold_list`, `case_list`, and `map_list`.

composite These are functions defined using the `def` syntax, as in the definition of `member`. They are essentially built up from the composition of other functions.

macro Macros are functions which are optionally defined as a kind of formal parameter for some composite function. They only exist within the scope of the function in which they are used, and are *not* first class objects. Macros enable the programmer to pass expressions to functions. One main use is that they allow the user to write polymorphic functions. For instance, in the definition of `member`, `eqFunc` is some test of whether two elements of an arbitrary type are the same (these two elements must be of the same type, and `eqFunc` must return a boolean, by the the *unification* algorithm in chapter 9). By making it a macro, the `member` function has been generalized to work over any type of list (eg. list of nats, list of bools,

etc.).

Each function must know to which kind it belongs, and must know the number of macros that it can take (this is always zero for macros and structors). In addition, each kind has a different set of data which must be stored in entries of its instances:

position *Macros* and *structors* must know the order in which they have been defined. This is so that Chirp and Charm can refer to them by indices rather than by name.

combinator-class & combinator-kind All functions except *macros* must know these two tags. They are not used by Chirp, but are of use to Charm.

signature All functions except *macros* have a *type signature* inferred by the type-checker.

macro-signatures All functions which have *type signatures* also have an array of type signatures for their *macros*, with the exception of *structors* (since they take no *macros*).

type-name *Structors* and *Ductives* know the name of the type they belong to.

structor-typing This is a type signature similar to *signature* defined above, but contains some special information needed by Charm (for doing its *substitutions*). It is not used by Chirp.

code *Composite* functions know the categorical combinator code that they compile into.

Variables

Entries of class *variable* have no special associated information. Unlike many other languages, the only information we need to know about an variable identifiers in Charity is that they are in fact variables. This is due to the manner in which the translation algorithm works (see chapter 8), and the fact that typechecking occurs *after* the translation process. One may think of variables more as place-holders.

Variables names have scope limited to the expression in which they are used (eg. a function body (as for `el` and `lst`, in figure 7.2), or an element of a fold operation (as for `el'` and `found`, in the same example)). Variables are declared in *variable bases* for a *terms* (expressions), and bind the occurrences of the variables therein.

Arbitrary Types and Type Representers

In figure 7.1, `A` is an *arbitrary type*, while `N` is a *type representer*. These have scope within a datatype definition only.

The only data we need to know about an arbitrary type is an integer representing its order within the list of arbitrary types about a datatype (for instance, `A` is the zeroth arbitrary type for `list`). This is so that Chirp and Charm may refer to these by their indices rather than their names.

We need no extra data regarding type representers—only what their names are.

7.2 Lexical Scoping

As suggested above, identifiers have a *lexical scope*. All primitive functions are defined at the lowest level (the zeroth level). A user's own function and type definitions (with the exceptions of macros) are inserted at level one. The scope of all other identifiers depends on their context.

The symbol table is implemented as a stack. As the parser directs the translation of the input source text, it *pushes* and *pops* new scopes on and old scopes off the stack, respectively. Any identifier who's entry is in an old scope ceases to exist (as far as the symbol table is concerned) once we move out of (pop) that scope.

For example, when we define `member` in figure 7.2, we first push a new scope onto the stack (ie. the symbol table). When we insert new entries (for `eqFunc`, `el`, and `lst`), they are added to that new, higher scope. For each element of the fold and case operations inside the function definition, we will again push and pop scopes, in turn. This causes `el'` and `found` to exist at an even higher level, for a briefer time. They

could temporarily overload different objects of the same name appearing outside the fold element in which they appear. Further, we could reuse the same identifiers to refer to other objects elsewhere in the function definition, or in the entire input source text.

Specifically, the symbol table is a stack of scopes, where each scope is an array of pointers to entries. We hash entries (by name) into a scope using external chaining. This allows fast lookup of entries, as well as the ability to overload identifier names. The stack implementation is also nice in that should Charity ever be extended to handle hierarchical function definitions, etc.[2] (as in Pascal, where the definitions of functions can be local to other functions), the symbol table will not need to be modified heavily, as the same general scoping rules could be enforced by pushing and poping scopes.

7.3 Interface Routines

Other Chirp modules need to be able to access the data stored in the symbol table, but in a controlled way. The following sections describe these interface routines.

7.3.1 Table Maintenance

The waiter needs some way of telling the symbol table to construct, reset, and destruct itself. Also, it needs some way of telling the symbol table to purge itself of all definitions made in the current working file (so that we may re-read it). Lastly, the parser needs to update the symbol table after completing the translation of the input source text.

`ConstructSystem()` Initialize the symbol table.

`DestructSystem()` Cleanup the symbol table.

`ResetSystem()` Cleanup, then re-initialize the symbol table.

`PurgeStab()` Traverse the symbol table, removing all entries while have been marked as being created in a file.

[2] See chapter 11.

`PostStab()` Traverse the symbol table. If an error has occurred during the translation, remove all entries which were probational. Otherwise mark them all as non-probational.

7.3.2 Inserting New Entries

The symbol table would be useless without allowing a routine for adding new entries. There are two functions:

`Insert()` This routine inserts data into the newest lexical scope.

`InsertAtBottom()` This is useful when we know we want to insert at the first scoping level (ie. when inserting structors).

Both these functions take a name, a class, and a union of the various different data we need, based upon the class. Not all data needs to be passed in (eg. the probational flag, etc.) as the symbol table can figure out such information for itself.

Note that information is often added to entries when they are inserted. However, we may initially insert data which is incomplete, then later look an entry back up and fill it in. For instance, we have no way of filling in type signatures for functions at definition time, since they must first be translated into categorical combinators.

7.3.3 Looking Up Data

If we can insert entries, we need to be able to look them up. We provide four routines, discussed below:

`LookUp()` See below.

`LookUpPrimitive()` See below.

`Find()` See below.

`FindAtBottom()` See below.

The two routines, `LookUp()` and `LookUpAtBottom()`, which are dual to `Insert()` and `InsertAtBottom()`, take a name as a key and lookup an entry with that name. In the case of `LookUp()`, we do not necessarily get something at the highest level, but rather we will get the entry which is as high as possible, if such an entry exists. Entries are not guaranteed to exist, and so these functions may return a null value as opposed to an entry. Procedures `Find()` and `FindAtBottom()`, however, *do* guarantee an entry. An error will be flagged otherwise.

7.3.4 Miscellaneous

The symbol table also provides the following routines:

`DumpTable()` This is for printing out all entries in the symbol table.

`PrintSubs()` This is for printing out a given "structor-typing" structure.

Chapter 8

Translating the Term Logic

The Charm abstract machine performs computations by evaluating streams of categorical combinators. These combinators can be thought of as "machine language instructions" for Charm: They are variable free, and a sequential list of them represents a program (assuming that it typechecks properly).

In theory, a programmer could write Charity code strictly at the combinator level, and Chirp in fact supports this[1]. However, like most machine language programming, this is difficult and hard to read. We would like to provide a high level language for writing Charity programs, which can then be compiled into combinators automatically. Such a language is provided—the **term logic**—and the vast bulk of Chirp is essentially the compiler which translates term logic source text into categorical combinators.

As Chirp parses the input source text, it builds up an intermediate representation of that input. This recursive structure is not so much a parse tree, as it is an error-free mirror of the logical structure of the *terms* in the input, and the *variable bases* upon which those terms are based. Chirp's **translator** module, then, takes this structure which has been built during parsing, and "flattens it out" into a steam of categorical combinators.

[1] With the `cdef` and `ceval` syntax, corresponding to defining composite functions in terms of combinators, and evaluating a stream of combinators, respectively.

8.1 Terms

Terms, which are essentially expressions, were introduced in chapter 2. As the parser sorts through the *syntactic* structure of the input source text, one of its main tasks is to build up an intermediate representational structure which embodies the form of the input source text. This structure is really just a recursive encoding for a term. When the parser has read in a complete function definition or expression, then if no errors have been detected, we know that the structure we are left with, together with the information we have added to the symbol table, is enough to call the `Translate()` routine in the translation module, which will return a list of categorical combinators.

The C code for the structure of a term is listed in figure 8.1, and the reader may verify that it indeed embodies all that we need to know about a term.

8.2 Variable Bases

Terms may contain *free variables*. That is, variables which appear in the term, and yet which are not declared within the term. Ultimately, however, any variable which appears must be *bound* by a variable in a variable base, either immediately before that term or in some nesting term.

A **variable base** is defined recursively as follows:

- $()$ is a variable base.
- if x is a variable, then x is a variable base.
- if v_0 and v_1 are variable bases, then (v_0, v_1) is a variable base.

For example, in figure 7.2, the variable `lst` appears exactly once in the body of the function definitions for `member`. Within the body (which is really just a term) then, `lst` is free. However, the formal parameter list for `member` (which is really just a variable base) contains `lst`, and so it has been properly bound (otherwise an error would result at parse time).

Just as the parser will build representations for *terms*, it also builds representations for *variable bases*. The C code for the structure of a variable base is listed in figure 8.2, and again the reader may verify that it embodies the nature of a variable base.

8.3 Abstracted Maps

We now know enough to fully characterize a Charity program: A **Charity program** is an **abstracted map** $\{v \mapsto t\}$ where v is a variable base containing all the free variables of the term t. Both v and t are represented within Chirp by the intermediate data structures described above. Chirp may then invoke its translation module (which takes the encoding for v and t as parameters, and returns a linear, variable-free stream of categorical combinators ready for typechecking and later evaluation).

8.4 Categorical Combinators

Categorical combinators, as stated above, are the variable-free "instructions" for the abstract machine, Charm, which evaluates Charity expressions. Producing them from the term logic input source text is the goal of the entire compilation process. Combinators are discussed in greater detail in chapters 2 and 10. Generally, combinator code is simply a string of functions, all composed together.

8.5 Translating Abstracted Maps to Combinators

The translation function $\mathcal{T}$ from abstracted maps in the term logic to categorical combinators is defined in tables 8.1 and 8.2.

$\mathcal{T}$ is $O(n \log n)$, which has been proven to be the lower bound on such a translation algorithm. $\mathcal{T}$ is a modified version of a translation function given in [4] and proven in [9]. The modifications, by Marc Schroeder, are the following:

- $\mathcal{T}[v \mapsto ()] =!$,

- $\mathcal{T}[x \mapsto x] = 1$,

- $\mathcal{T}[(v_0, v_1) \mapsto x] = \mathrm{p}_i; \mathcal{T}[v_i \mapsto x]$where $i = 0$ if x occurs in v_0 otherwise $i = 1$,

- $\mathcal{T}[v \mapsto (t_0, t_1)] = \langle \mathcal{T}[v \mapsto t_0], \mathcal{T}[v \mapsto t_1] \rangle$,

- $\mathcal{T}[v \mapsto f(t)] = \mathcal{T}[v \mapsto t]; f$,

- $\mathcal{T}[v \mapsto f\{(v_1 \mapsto t_1), ..., (v_n \mapsto t_n)\}(t)]$
  $= \langle \mathcal{T}[v \mapsto t, 1] \rangle; f\{\mathcal{T}[(v_1, v) \mapsto t_1], ..., \mathcal{T}[(v_n, v) \mapsto t_n]\}$,

- $\mathcal{T}[v \mapsto \{w \mapsto t'\}(t)] = \langle \mathcal{T}[v \mapsto t], 1 \rangle; \mathcal{T}[(w, v) \mapsto t']$,

Table 8.1: The first set of translations for the term logic.

- $\mathcal{T}\left[v \mapsto \left\{\begin{array}{ccc} \mathsf{c}_1(v_1) & \mapsto & t_1 \\ & \vdots & \\ \mathsf{c}_n(v_n) & \mapsto & t_n \end{array}\right\}(t)\right]$
  $= \langle \mathcal{T}[v \mapsto t], 1\rangle; \mathsf{case}^L\{\mathcal{T}[(v_1, v) \mapsto t_1], ..., \mathcal{T}[(v_n, v) \mapsto t_n]\},$

- $\mathcal{T}\left[v \mapsto \left\{\!\!\left\{\begin{array}{ccc} \mathsf{c}_1 : v_1 & \mapsto & t_1 \\ & \vdots & \\ \mathsf{c}_n : v_n & \mapsto & t_n \end{array}\right\}\!\!\right\}(t)\right]$
  $= \langle \mathcal{T}[v \mapsto t], 1\rangle; \mathsf{fold}^L\{\mathcal{T}[(v_1, v) \mapsto t_1], ..., \mathcal{T}[(v_n, v) \mapsto t_n]\},$

- $\mathcal{T}\left[v \mapsto L\left\{\begin{array}{ccc} v_1 & \mapsto & t_1 \\ & \vdots & \\ v_m & \mapsto & t_m \end{array}\right\}(t)\right]$
  $= \langle \mathcal{T}[v \mapsto t], 1\rangle; \mathsf{map}^L\{\mathcal{T}[(v_1, v) \mapsto t_1], ..., \mathcal{T}[(v_m, v) \mapsto t_m]\},$

- $\mathcal{T}\left[v \mapsto \left(\!\!\left| w \mapsto \begin{array}{c} \mathsf{d}_1 : t_1 \\ \vdots \\ \mathsf{d}_m : t_m \end{array} \right|\!\!\right)(t)\right]$
  $= \langle \mathcal{T}[v \mapsto t], 1\rangle; \mathsf{unfold}^L\{\mathcal{T}[(v, w) \mapsto t_1], ..., \mathcal{T}[(v, w) \mapsto t_n]\},$

- $\mathcal{T}\left[v \mapsto \left(\begin{array}{c} \mathsf{d}_1 : t_1 \\ \vdots \\ \mathsf{d}_m : t_m \end{array}\right)\right] = \langle !, 1\rangle; \mathsf{record}^L\{\mathcal{T}[v \mapsto t_1], ..., \mathcal{T}[v \mapsto t_n]\}.$

Table 8.2: The second set of translations for the term logic.

- Addition of a rule for translating functions taking macros, which was omitted in the referenced works.
- Correction of the rule for records, which was missing the first combinator (the pair combinator) on the right hand side.
- The generalization of a rule for translating functions, which was initially subdivided into separate rules for differing kinds of functions, in the referenced works.

8.6 A Quick Example

For illustrative purposes, consider the following simple Charity program for adding two numbers:

```
% define the natural numbers:

data nat -> N =
       zero : 1 -> N
     | succ : N -> N.

% the actual function which does the adding:

def add(x, y) =
     {| zero : ()     => x
      | succ : (sum) => succ(sum)
      |}
         (y).
```

When actually entered and compiled, the resulting categorical combinator code produced by Chirp was:

```
<p1;id,id>;fold_nat{p1;p0;id,p0;id;succ}
```

It does not take long to apply $\mathcal{T}$ by hand, and check that the result given by Chirp is in fact correct.

8.7 Interface Routines

The translation module provides the following services to other Chirp modules:

`Translate()` For converting abstracted maps to categorical combinators, as described above.

`DestructVBase()` For cleaning up variable base intermediary representational structures once they have been used.

`DestructTerms()` As `DestructVBase()`, but for *terms*.

`DestructCombs()` For cleaning up a stream of categorical combinator code when we are done with it (usually after an expression has been evaluated, or when we destruct the symbol table).

`PrintCode()` A routine for printing out categorical combinator code in human-readable form.

`PrintCodeAsLogic` A routine that converts categorical combinator code back into the term logic (or a textual representation of the term logic, at least) for displaying to the user (ie. to show the results of an evaluation).

Note that `Translate()` is the only routine in the translation module, accessible to other modules, which is directly involved with translation. However, the other routines are present in that module since it has the most to do with forms of representation of categorical combinators and term logic.

```
typedef enum {
  R_EMPTY = 0,
  R_VARIABLE,
  R_ABSTRACTION,
  R_FUNCTION,
  R_PAIR,
  R_CASE,
  R_FOLD,
  R_MAP,
  R_UNFOLD,
  R_RECORD
} termKind;

typedef struct {
  varBase      *v;
  struct _term *t;
} CFMAEl;

typedef struct _term {
  termKind kind;

  union {
    char *x;

    struct {
      varBase      *w;
      struct _term *t_;
      struct _term *t;
    } abstraction;

    struct {
      stabEntry    *func;
      CFMAEl       *abstr;
      struct _term *t;
    } function;

    struct {
      struct _term *t0;
      struct _term *t1;
    } pair;

    struct {
      char         *type;
      CFMAEl       *abstr;
      struct _term *t;
    } caseOrFoldOrMap;

    struct {
      char         *type;
      struct _term **abstr;
      varBase      *w;
      struct _term *t;
    } unfold;

    struct {
      char         *type;
      struct _term **abstr;
    } record;
  } data;
} term;
```

Figure 8.1: The source code for representing Charity terms.

```
typedef enum {
  B_EMPTY,
  B_VARIABLE,
  B_PAIR
} baseKind;

typedef struct _varBase {
  baseKind kind;

  union {
    char *x;

    struct {
      struct _varBase *v0;
      struct _varBase *v1;
    } pair;
  } data;
} varBase;
```

Figure 8.2: The source code for representing Charity variable bases.

Chapter 9

Typechecking

A few semantic checks are performed by the system at parse-time. Strangely enough, however, the main thrust of the semantic checking comes *after* input source text has been parsed, and *even after* it has been compiled. Once we have a stream of categorical combinators, Chirp applies its typechecking algorithm—a *unification* algorithm. The purpose of this is twofold:

- To ensure that the input is not nonsensical, by testing that it types properly.
- To generate **type signatures** for functions.

9.1 The Nature of Typechecking in Charity

Charity is a *statically* and *strongly* typed language. This means that all typechecking is performed at *compile* time, and that the system checks to ensure that all interrelated functions and expressions have typings which "mesh together" (or *unify*) properly. Strong typing is proving very useful in program development, as it helps ensure that when a program compiles with the expected typing, there is better evidence that it is in fact correct (compare C or Lisp).

Still, the typechecker only tries to unify in the most general way. This allows the definition of functions which are **polymorphic**. That is, the **type signatures** that are inferred for each function may contain arbitrary types, so that the function can be called with varying input types, and consequently may produce varying output types. This is powerful, since it allows the creation of generalized functions (recall the definition of the `member` function in figure 7.2).

9.2 The Theory Behind the Typechecker

Until now, we have referred to categorical combinators as some kind of "machine instructions" for the abstract machine, Charm. They have an interpretation on a theoretical level as well: They are *morphisms* in a category. Recall from chapter 2 that in categorical programming, the developer is working within a category—*objects* are datatypes, and the *morphisms* between objects are functions (or programs) which map an input datatype to an output datatype.

It should now become apparent that the typeckecking algorithm will really only be checking that the codomain of some combinator unifies with the domain of the next combinator in the sequence.

A sequence of categorical combinators presented as an expression or as the code for a new function is really, then, just the composition of morphisms in a category. Since categories are closed under composition, then as long as all the combinators unify with those on either side in the sequence, the composition of all the combinators is guaranteed to be some new, valid morphism in the category (in fact, it is Charm's goal to rewrite the sequence of combinators, as much as possible, into that more simplified morphism—see chapters 2 and 10).

9.3 The Unification Algorithm

We are now almost ready to give the unification algorithm. We must note three final details:

- The result of performing the unification will not only be that we know if the code is semantically valid, but also that we will have produced the typing for the morphism which is the composition of the sequence of combinators. This is a **type signature**.
- As noted previously, Charity allows functions to take macros. This means that combinators may also have macros. But macros will simply translate into sequences of combinators themselves (see the translation algorithm $\mathcal{T}$ in chapter 8 for the details). Thus, typechecking a sequence of combinators implies that we must also recursively typecheck their macros.
- If we are type checking a function which takes macros, we must also generate the type signatures for those macros. Thus, when we encounter a macro for the function as a combinator in the sequence, we must lookup its typing, unify it with the surrounding combinators, and store that updated typing back for future reference (with other combinators, we can simply discard the results of the unification, as their types signatures have already been fully defined).

The unification algorithm is now given in figure 9.1.

There are some subtle points to consider. The array M is the table of signatures for all macros M_i for the function we are typechecking. Initially they are as general as possible, since their signatures have not been developed through unification. By the end of the process, however, they will likely have been refined. M and m should be considered global to all algorithms presented here. Further, calls to UNIFY pass the *resultant* signature by reference. This is because *resultant* is the type we want to update as the result of unifying. It is for this reason that when we lookup a type signature for some previously defined function, we must make a copy of it. Also, the arbitrary types

in the signatures for new copies must all be new (ie. they can't have been used as the names for arbitrary types in other signatures already).

9.4 Type Signatures

The type signatures produced by the unification algorithm are not needed when the code corresponds to some expression to be evaluated. However, when it corresponds to a function definition, the signature (and potentially the type signatures for macros of that function) must be kept. This is because we may need to refer to that type signature again later in subsequent typechecking operations, if we build a new composite function from the old one.

The typechecker knows when it is working on a function definition, and so will automatically fill out the relevant type signatures in the symbol table entry for the given function (see section 7.1.1).

9.5 Interface Routines

The typechecking module provides the following interface routines:

`TypeCheck()` This is the procedure which accepts code to typecheck, performs the unification, generates a type signature, etc., as described above.

`CopyType()` This procedure mades a duplicate of a type signature. It is mainly useful outside of the typechecker only when constructing the symbol table.

`DestructType()` This routine cleans up typing information, when an associated symbol table entry is being destroyed.

`PrintType()` This procedure traverses a type signature and prints it out in human-readable form.

ALGORITHM TYPECHECK(s)

```
if we are typechecking a new function definition
then
  let m = the number of macros for the function f we are typechecking;
  let M be an array of macro type signatures of size m;
  let M_i = α_i ↦ β_i be a unique arbitrary typing for each macro i, 1 ≤ i ≤ m, of f, initially;
  let result = TYPECHECK′(s);
  update type signature and macro type signatures for f with result and M, respectively;
  return result;
else
  return TYPECHECK′(s);
```

Figure 9.1: The typecking algorithm employed by Chirp.

ALGORITHM TYPECHECK′(s)

```
let s be a sequence of combinators;
  case s of
    composition of subsequences, s = s_1; s_2:
      return UNIFYMAPS(TYPECHECK′(s_1), TYPECHECK′(s_2));
    combinator, s = c:
      return GETTYPESIGNATURE(c);
    macro, s = M_i, 1 ≤ i ≤ m:
      return M_i;
```

Figure 9.2: The subordinate typecking algorithm.

GETTYPESIGNATURE(c)

```
let t = lookup of type signature for c;
let u be a copy of t, ie. all arbitrary types in u are new;
let u take v macros;
foreach macro type signature t_w, 1 ≤ w ≤ v for c do:
  let m_w be the combinator code for macro w in c;
  UNIFY(TYPECHECK(m_w), t_w, u);
return u;
```

Figure 9.3: The algorithm for looking up a type signature.

ALGORITHM UNIFYMAPS(f_1, f_2)

```
let d_1 be the domain of f_1;
let c_1 be the codomain of f_1;
let d_2 be the domain of f_2;
let c_2 be the codomain of f_2;
let signature initially be the new type d_1 ↦ c_2;
UNIFY(c_1, d_2, signature);
if ISMACRO(f_1)
  UNIFY(c_1, d_2, f_1);
if ISMACRO(f_2)
  UNIFY(c_1, d_2, f_2);
return signature;
```

Figure 9.4: The algorithm to unify two maps.

ALGORITHM UNIFY(t_1, t_2, *resultant*)

```
case t1 of
  non-arbitrary type A(a1, ..., ak):
    case t2 of
      non-arbitrary type B(b1, ..., bl):
        if A ≠ B then error else
          for c ← 1 to k do:
            UNIFY(ac, bc, resultant);
      arbitrary type B:
        if B occurs in A(a1, ..., ak) then error else
          rewrite B with A(a1, ..., ak) in resultant;
  arbitrary type A:
    case t2 of
      non-arbitrary type B(b1, ..., bl):
        if A occurs in B(b1, ..., bl) then error else
          rewrite A with B(b1, ..., bl) in resultant;
      arbitrary type B:
        rewrite A with B in resultant;
```

Figure 9.5: The unification algorithm employed by Chirp.

Chapter 10

The Chirp/Charm Interface

As already stated, Chirp is simply one part (the front end) of the larger interpreter. It is responsible for:

- Interacting with the user (by responding to commands, allowing the user to make data and function definitions, handling requests for the evaluation of expressions, etc.).
- Scanning the input, parsing it, checking its semantics, and translating it.

However, all this would be useless if not for the ability to evaluate expressions. This is not the domain of Chirp, but of Charm.

The version of Charm which Chirp interfaces with has been written by Barry Yee [23], but much of the original design was done by Mike Hermann [15].

This chapter will give an overview of Charm, and explain how both ends of the interpreter interact. The shared data structures, as well as the connecting procedures, will be explained.

10.1 What is Charm? An Overview:

Charm is a categorical abstract machine which takes a list of categorical combinators and rewrites them into normal form. This process is guaranteed to terminate. The theory behind this is presented in chapter 2.

Charm implements strategies for taking computational "shortcuts", when possible, to speed up evaluation of expressions. This is known as **lazy evaluation**, and it is implemented through *graph reduction* and *sharing*. Lazy evaluation is one of the best arguments for using modern interpreted functional languages.

10.2 Shared Data Structures

The principle data communicated between Chirp and Charm is sequences of categorical combinators. The data structure used by the system is given in figure 10.1. Data of this type is passed in to Charm before an evaluation can take place, and data of this type is passed back out as the result.

A combinator is essentially the name of a morphism, along with an optional list of macros, where each macro is the composition of more combinators.

As explained in chapter 7, constructors and destructors have special associated typing information (in addition to their type signatures). This is information needed by Charm in order to perform *substitutions*. It has little to do with Chirp besides the fact that Chirp must build this data up, and make it available to Charm. The reader is referred to [23] for more information. The data structure for representing the substitution information is presented in figure 10.2.

10.3 Invoking the Back End

There are two routines by which Chirp invokes Charm:

`Compile()` This procedure passes Charm some combinator code, together with a unique name for that code. It initializes Charm's own internal tables, and asso-

```
typedef enum {
  Comp,
  Comb,
  Parm,
  Unpr             /* unused */
} CTERM_TYPE;

typedef enum {
  Active,
  Ractive,
  Inactive,
  Dist,
  Cdefs
} COMB_CLASS;

typedef enum {
  CK_CASE,
  CK_FOLD,
  CK_UNFOLD,
  CK_RECORD,
  CK_MAP_L,
  CK_MAP_R,
  CK_OTHER
} combKind;

typedef struct Cterm_node *CTERM;

typedef struct Cterm_list_node *CTERM_LIST;

typedef struct Cterm_list_node {
  CTERM      cterm;
  CTERM_LIST next;
} CTERM_LIST_REC;

typedef struct Cterm_node {
  CTERM_TYPE tag;

  union {
    struct {
      CTERM cterm1;
      CTERM cterm2;
    } composition;

    struct {
      char       *name;
      COMB_CLASS class;
      combKind   kind;
      CTERM_LIST cterm_list;
    } combinator;

    int position;
  } type;
} COMB_TERM;
```

Figure 10.1: The source code for representing categorical combinators.

```
typedef enum {
  ST_NONREC,
  ST_REC
} subType;

typedef enum {
  F_1,
  F_ID,
  F_PROD,
  F_OTHER
} factorizer;

typedef struct _substitution {
  factorizer kind;

  union {
    struct {
      subType arg;
      int     posn;
      char    *type;
    } id;

    struct {
      struct _substitution *s1;
      struct _substitution *s2;
    } prod;

    struct {
      char                 *type;
      int                  numSubs;
      struct _substitution **s;
    } other;
  } data;
} substitution;
```

Figure 10.2: The source code for representing substitution information.

ciates the name with the sequence of combinators. It does *not* actually evaluate anything. If the code corresponds to a function definition, nothing else needs to be done. We simply call `Compile()` after the translation and typechecking has been accomplished, and we are done. However, if the code is an expression, we immediately follow this call with a call to `Machine()`.

`Machine()` This procedure evaluates the code with the given name (that association would have been made by a previous `Compile()` call). It returns the combinator code for the normal form of the named expression.

10.4 Querying the Front End

As Charm carries out is computation, it needs to query Chirp's tables for essential information. It requires the following routines (again, the reader is referred to [23]), which simply provide a clean interface to the symbol table:

`SubLookup()` This procedure returns the substitution data for a given constructor or destructor.

`StructorPosn()` This procedure returns an integer corresponding to a constructor or destructor's position in the definition for its datatype.

`CombsFor()` This procedure returns an array of the names of the constructors or destructors for a given type.

Chapter 11

Future Work

Charity, as a language, is not yet completely developed, as there are many features the various people involved would like to see included[1]. As Charity, and categorical programming in general, are still new and active research fields, there is also work to be done on some theoretical issues behind the language[2]. Similarly, then, one should not consider work on Chirp to be complete.

This chapter will list some suggestions (some almost trivial to incorporate, but valuable, and others more challenging) for the extension of Chirp. It is thus hoped that Chirp will continue to develop as a useful, powerful tool for the modern programmer. We will also touch upon some ideas for the development of the Charity language itself.

11.1 Extending Chirp

Chirp, like any large system, is never really complete. Some ideas for the future extension of the system includes:

[1] For instance, the subject of Tom Fukushima's PhD thesis is adding *monads* to Charity. Don Kuzenko is working on pattern matching.

[2] Proof theory, for instance.

- Allowing the user to pre-specify type signatures for functions. This would allow for a limited about of self error checking as well as the ability to contrain signatures which would otherwise be more polymorphic.
- The ability to remove function definitions. This implies the existence of a dependecy checker.

11.2 Extending Charity

The Chirp proposal[17] listed some possible extensions to the Charity language itself. Others have been suggested since development of the Chirp system was undertaken. *Some* of these are described below:

The major extension to be considered was adding input/output. After surveying the literature, however, it was discovered that this would be quite a large task, perhaps the subject of a masters thesis. One main reason for the difficulty is that to maintain the nice proof properties for programs in Charity, no side effects are allowable [13][16]. This means that the simplest form of I/O implementable, *side-effected I/O*, is not a consideration (as it has been for most other languages). It is believed that I/O should still be added to the language at a later date.

Robin Cockett believes that a front end for the parser should be added which allows the user to define syntax extensions. For this reason, hard coded extended syntactic forms have been left out of Chirp.

Bibliography

[1] AV Aho, R Sethi, JD Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] M Barr, C Wells. *Category Theory for Computer Science*. Prentice Hall, 1990.

[3] R Bird, P Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[4] JRB Cockett, T Fukushima. *About Charity* (draft manuscript). University of Calgary, Feb. 1992.

[5] JRB Cockett, HG Chen. *Categorical Combinators*. Submitted to 3rd Banff Higher Order Workshop.

[6] JRB Cockett. *Notes on Strength, Datatypes, and Shape* (draft manuscript). Macquarie University, May 1991.

[7] JRB Cockett. *12 Lectures on Categorical programming* (lecture notes). Sydney University, Sept-July 1990.

[8] JRB Cockett, D Spencer. Strong Categorical Datatypes I. In RAG Seely, editor, *International Meeting on Category Theory 1991, Canadian Mathematical Society Proceedings*. AMS, Montreal, 1992.

[9] JRB Cockett, D Spencer. *Strong Categorical Datatypes II: A Term Logic for Categorical Programming* (draft manuscript). May 1992.

[10] JRB Cockett. Conditional Control is not Quite Categorical Control. In G Birtwistle, editor, *IV Higher Order Workshop, Banff, 1990*. Springer-Verlag, Workshops in Computing, 1991, pp. 190-217.

[11] P Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming*. Pitman Wiley, 1986.

[12] T Fukushima. *Charity User Manual* (draft manuscript). University of Calgary, Nov. 1991.

[13] AD Gordon. *Functional Programming and Input/Output* (PhD Thesis). University of Cambridge, August 1992.

[14] T Hagino. *A Categorical Programming Language* (PhD thesis). University of Edinburgh, 1987.

[15] M Hermann. *A Lazy Graph Reduction Machine for Charity: Charity Abstract Reduction Machine (CHARM)* (draft manuscript). University of Calgary, Jul. 1992.

[16] P Hudak, RS Sundaresh. *On the Expressiveness of Purely Functional I/O Systems*. Yale University, March 1989.

[17] M Schroeder. *Proposal: Chirp: A Front End for Charity*. University of Calgary, Oct. 1992.

[18] D Spencer. A Survey of Categorical Computation: Fixed Points, Partiality, Combinators, . . . Control?. *Bulletin of the European Association of Theoretical Computer Science 43*. Feb. 1991, pp. 285-312.

[19] P Taylor. *Recursive Domains, Indexed Category Theory, and Polymorphism* (PhD thesis). University of Cambridge, 1986.

[20] RFC Walters. *Categories and Computer Science*. Carslaw Publications, Sydney, 1991.

[21] RFC Walters. Datatypes in Distributive Categories. *Bulletin of the Australian Mathematical Society 40*. 1989, pp. 79-82.

[22] GC Wraith. A Note on Categorical Datatypes. *Lecture Notes in Computer Science 389*. Springer-Verlag, 1989, pp. 118-127.

[23] B Yee. *Charm*. CPSC 502 project report.

Appendix A

Lexical Units

This appendix details how Chirp groups input source text into tokens, for subsequent parsing. There are tokens for each reserved word and each symbol, as well as one token for identifiers, one for commands, and ond for the "end of file" condition.

A.1 Reserved Words

- `data`
- `def`
- `cdef`
- `ceval`

A.2 Symbols

```
.    (   )
{    }   ->
=>   =   :
|    *   +
,    -   ;
!    1
```

A.3 Identifiers

Identifiers have the following form: Each identifier begins with an upper or lower-case alphabetic character. This initial character may be followed by an arbitrary number of the following characters: another alphabetic character (either upper or lower-case), a numeric digit, an underscore (_), or an apostrophe ('). An identifier ends with the first character not of this form. Note that Charity is *case sensitive*.

A.4 Miscellaneous

There is a "command token", which denotes that the user has issued a command to the interpreter. This token has an associated data field indicating which command has been given. See appendix D for a list. Commands begin with a slash (/), followed by at least as many characters of the command name as the user needs to type in order to disambiguate it from other commands (note that the (/) convention cuts down on the needed keywords in the language, and distinguishes commands to the system *itself* from language constructs).

There is also a special token which represents the "end of file" condition, which is only encountered when reading input from a file (an "end of file" condition met during keyboard input is interpreted as a request to end the current Charity session).

Tabs, *spaces*, and *newlines* are "white space". Comments are also treated as white

space. They may either be enclosed within the delimiters `(*` and `*)`, in which case they may span multiple lines, or they may start with `%`, in which case they end when a newline is encountered. *Comments may be nested.*

Appendix B

Grammar

This appendix gives the left-recursionless BNF grammar for Charity, used by Chirp's "near-predictive" parser. A few productions have not been left-factored (for clarity), but could easily be.

Productions which cause the grammar to fail to be purely *LL(1)* are marked with †.

This grammar was written by Marc Schroeder, but embodies the syntactic structure of the Chirp prototype written by Tom Fukushima, under Dr. Robin Cockett.

Conventions obeyed in the grammar below are as follows:

- The start symbols are labeled *ITEM*.
- Other non-terminals are labeled *item*.
- Reserved words are labeled `item`.
- Symbolic tokens are labeled "`item`"
- The "end of file" token is labeled specially as **eof**.
- Other terminals (ie. ones with attributes—identifiers and commands) are labeled **`item`**.
- The empty string is denoted by ϵ.

FILE →
    elementList **eof** .

elementList →
    element elementList
  | ϵ .

ADHOC† →
    element
  | *generalCommand* "`.`" .

element →
    dataDef "`.`"
  | *functionDef* "`.`"
  | *expression* "`.`"
  | *cDef* "`.`"
  | *cExpression* "`.`"
  | *specialCommand* "`.`"
  | "`.`" .

dataDef† →
    inductiveDef
  | *coinductiveDef* .

inductiveDef →
    `data` **`identifier`** *finiteProd* "`->`" **`identifier`** "`=`" *constructorList* .

finiteProd →
    "`(`" *identList* "`)`"
  | ϵ .

identList →
    `identifier` *identList′*
  | ϵ .

identList′ →
    "`,`" **`identifier`** *identList′*
  | ϵ .

constructorList →
    constructor constructorList′ .

constructorList′ →
    "`|`" *constructor constructorList′*

| ϵ .

constructor →
identifier "`:`" *type* "`->`" **identifier** .

type →
type′
| *type′* "`*`" *type′*
| *type′* "`+`" *type′* .

type′ →
"`(`" *type* "`)`"
| "`1`"
| **identifier**
| **identifier** "`(`" *identList* "`)`" .

coinductiveDef →
`data` **identifier** "`->`" **identifier** *finiteProd* "`=`" *destructorList* .

destructorList →
destructor destructorList′ .

destructorList′ →
"`|`" *destructor destructorList′*
| ϵ .

destructor →
identifier "`:`" **identifier** "`->`" *type* .

functionDef →
`def` **identifier** *formalMacroList formalParamList* "`=`" *expression* .

formalMacroList →
"`{`" *identList* "`}`"
| ϵ .

formalParamList →
pattern
| ϵ .

pattern →
"`(`" *pattern′* "`)`"
| *pElement* .

pattern′ →
    pElement
  | *pElement* “`,`” *pElement*
  | *pElement* “`,`” “`(`” *pattern′* “`)`”
  | “`(`” *pattern′* “`)`” “`,`” *pElement*
  | “`(`” *pattern′* “`)`” “`,`” “`(`” *pattern′* “`)`”
  | ϵ .

pElement →
    `identifier`
  | “`_`” .

expression →
    functionOrVarOrMap
  | *foldOrCaseOrAbs*
  | *unfoldOrRecordOrOther* .

closedExpression →
    “`(`” *closedExpression′* “`)`” .

closedExpression′ →
    expression possibleSecond
  | ϵ .

possibleSecond →
    “`,`” *expression*
  | ϵ .

functionOrVarOrMap† →
    function
  | *variable*
  | *map* .

function →
    `identifier` *actualMacroList actualParamList* .

actualMacroList →
    “{” *abstractionList* “}”
  | ϵ .

abstractionList →
    abstraction abstractionList′
  | ϵ .

abstractionList′ →
    “`,`” *abstraction abstractionList′*
  | ϵ .

abstraction →
    pattern “=>” *expression* .

actualParamList →
    closedExpression
  | ϵ .

variable →
    `identifier` .

map →
    `identifier` “{” *mapElementList* “}” *closedExpression* .

mapElementList →
    abstraction abstractionList′ .

foldOrCaseOrAbs →
    “{” *foldOrCaseOrAbs′* “}” *closedExpression* .

foldOrCaseOrAbs′† →
    “|” *fold*
  | *case*
  | *abstraction* .

fold →
    foldElement “|” *foldElementList* .

foldElementList →
    foldElement “|” *foldElementList*
  | ϵ .

foldElement →
    `identifier` “`:`” *pattern* “=>” *expression* .

case →
    caseElement caseElementList .

caseElementList →
    “|” *caseElement caseElementList*
  | ϵ .

caseElement →
    identifier *possiblePattern* "=>" *expression* .

possiblePattern →
    pattern
  | ϵ .

unfoldOrRecordOrOther →
    "(" *unfoldOrRecordOrOther′* .

unfoldOrRecordOrOther′† →
    unfold
  | *record* ")"
  | ")"
  | *closedExpression′* ")" .

unfold →
    "|" *pattern* "=>" *unfoldElementList* ")" *closedExpression* .

unfoldElementList →
    unfoldElement "|" *unfoldElementList′* .

unfoldElementList′ →
    unfoldElement "|" *unfoldElementList′*
  | ϵ .

unfoldElement →
    identifier ":" *expression* .

record →
    recordElement recordElementList .

recordElementList →
    "," *recordElement recordElementList*
  | ϵ .

recordElement →
    identifier ":" *expression* .

cExpression →
    `ceval` *combinatorList* .

combinatorList →

combinator combinatorList′ .

combinatorList′ →
“**;**” *combinator combinatorList′*
| ϵ .

combinator →
combinatorName combParamList .

combinatorName →
identifier
| “!” .

combParamList →
“{” *combParams* “}”
| ϵ .

combParams →
combinatorList combParams′ .

combParams′ →
“**,**” *combinatorList combParams′*
| ϵ .

cDef →
`cdef` **identifier** *formalMacroList* “=” *combinatorList* .

specialCommand →
command .

generalCommand →
command .

Appendix C

Primitives

Here we list and describe all builtin primitive Charity combinators and types.

Primitive:	Kind:	Typing:
!	comb.	$\alpha \to 1$
id	comb.	$\alpha \to \alpha$
1	type	n/a
map_1	comb.	$1 \times \alpha \to 1$
$\to$ (map)	type	n/a
$\times$ (prod)	type	n/a
pair	comb.	$\{\alpha \to \beta, \alpha \to \gamma\} : \alpha \to \beta \times \gamma$
p0	comb.	$\alpha \times \beta \to \alpha$
p1	comb.	$\alpha \times \beta \to \beta$
map_prod	comb.	$\{\alpha \to \gamma, \beta \to \delta\} : \alpha \times \beta \to \gamma \times \delta$
$+$ (coprod)	type	n/a
b0	comb.	$\alpha \to \alpha + \beta$
b1	comb.	$\beta \to \alpha + \beta$
case_coprod	comb.	$\{\alpha \to \gamma, \beta \to \gamma\} : \alpha + \beta \to \gamma$
map_coprod	comb.	$\{\alpha \to \alpha, \beta \to \beta\} : \alpha + \beta \to \alpha + \beta$

Appendix D

Commands

This chapter lists all Chirp commands currently implemented, as well as some which may be incorporated in the near future[1].

Note that some commands may appear within input files *and* standard-input input source text, while others are valid *only from the Chirp prompt*.

`/dump` Display all defined types and combinators (command line only).

`/edit (filename)` Edit the named file. If no name is given, use the current working file (command line only).

`/file (filename)` Read in an input source file. If we are at the command line, the named file is the new current working file. If no name is given on the command line, we force a re-read of the current working file. working file

`/help` Display the *help* screen (command line only).

`/info` Display the *info* screen (command line only).

`/quit` Quit the system (command line only).

[1] Chirp also handles the old `readfile` and `restart_base` commands, for backwards compatibility.

`/reset` Reset the entire system (command line only).

`/query` Lookup inf

Appendix E

Example Script

An example Chirp session is listed here as a demonstration of what interaction with the system is like. It may also prove of limited usefulness in learning more about how to program in Charity.

```
*** Script initiated by Marc Schroeder on Fri Apr  2 14:05:58 1993
marc has logged on console from local.
marc has logged on ttyp0 from :0.0.
marc has logged on ttyp1 from :0.0.
marc has logged on ttyp2 from :0.0.
marc has logged on ttyp3 from :0.0.
[101 fo-ttyp4 2:06pm report] > ch

CHIRP - the CHarity InteRPreter.

( chirp v0.1 beta
  charm v0.1 beta )

Type  "/help"  for a list of OPTIONS,
type  "/info"  for basic system INFORMATION.

Using file: scrat.ch

Charity ready.
> /dump

-- top level --



-- lower level --

! 1 pair case_coprod map_prod prod map_1 map_coprod b0 b1 map_id id map coprod
p0 p1

> data nat -> N =
        zero : 1 -> N
      | succ : N -> N.
> def add(x, y) =
      {| zero : ()  => x
       | succ : sum => succ(sum)
       |}
          (y).
> /dump

-- top level --
```

```
add nat case_nat succ fold_nat zero

-- lower level --

! 1 pair case_coprod map_prod prod map_1 map_coprod b0 b1 map_id id map coprod
p0 p1

> add(zero, zero).
zero
> add(succ(succ(zero)), succ(succ(succ(zero)))).
succ(succ(succ(succ(succ(zero)))))
> /quit
Exiting Charity.

[102 fo-ttyp4 2:07pm report] > logout

*** Script completed on Fri Apr  2 14:07:53 1993
*** Script session length is 60 lines
```

Appendix F

Header Files

The header files for each Chirp module are listed in this appendix. They are given so that the curious reader may more closely scrutinize some (but not all) of the more important data structures in Chirp, as well as how the various modules in the system interact via the interface routines.

This section is mainly included for reference purposes, and is not intended as a description of how the system operates. These files are subject to change in subsequent releases.

F.1 `chirp.h`

This file is not so important to `chirp.c` (the waiter), as it is to the entire system. It makes definitions which will be of general use to all modules.

```
#ifndef __CHIRP__
#define __CHIRP__

#include <stdio.h>
#include <malloc.h>

#define CHIRP_VER "v0.1 beta"
#define private   static
#define EOS       '\0'

typedef enum {
  false = 0,
  true
} boolean;
```

```
#include "error.h"

#define mmalloc(ptr, type)        if ((ptr = (type *)malloc(sizeof(type)))
                                  == (type *)NULL) Panic(false, "malloc call")
#define mmallocn(ptr, type, n)    if ((ptr = (type **)
                                           malloc(sizeof(type *)* n))
                                  == (type **)NULL) Panic(false, "malloc call")
#define mmallocn2(ptr, type, n)   if ((ptr = (type *)
                                           malloc(sizeof(type) * n))
                                  == (type *)NULL) Panic(false, "malloc call");
#define mmallocx(ptr, type, size) if ((ptr = (type)
                                           malloc(size))
                                  == (type)NULL) Panic(false, "malloc call");

#endif
```

F.2 `defaults.h`

There is no corresponding code module to this file. Rather, it groups various system defaults into one location, for easy modification by someone setting up a working interpreter.

```
#ifndef __DEFAULTS__
#define __DEFAULTS__

#define EDITOR "mg"
#define PAGER  "more"

#define HELPFILE "/home/c502/L01/marc/charity/help"
#define INFOFILE "/home/c502/L01/marc/charity/info"

#endif
```

F.3 `parse.h`

This file describes the interface to the parser: code module `parse.c`.

```
#ifndef __PARSE__
#define __PARSE__

#include <setjmp.h>
#include "chirp.h"

typedef enum {
  A_NONE,
  A_EDIT,
  A_QUIT,
  A_RESET
} action;

extern action  Parse (char *fromFile, boolean lazy);
extern void    Lock  (void *ptr);

extern jmp_buf env;

#endif
```

F.4 `error.h`

This file describes the interface to the error hander: code module `error.c`.

```
#ifndef __ERROR__
#define __ERROR__

#include "chirp.h"

#define UNDEF_LINE 0

typedef enum {
  E_FATAL,
  E_ERROR,
  E_WARNING
} errorKind;

extern void    PreError         (void);
extern void    PostError        (void);
extern boolean Errors           (void);
extern void    ReportError      (errorKind kind, int errorLine, char *message);
extern void    ReportBadCommand (void);
extern void    _Panic           (boolean firewall, char *message, char *file,
                                 int errorLine);

#define Panic(firewall, message) _Panic(firewall, message, __FILE__, __LINE__)

#endif
```

F.5 `lex.h`

This file describes the interface to the lexical analyzer: code module `lex.c`.

```
#ifndef __LEX__
#define __LEX__

#include "chirp.h"

typedef enum {
  T_DUMMY = 0,
  T_DATA,
  T_DEF,
  T_CDEF,
  T_CEVAL,
  T_COMMAND,
  T_IDENT,
  T_DONE,
  T_LPAREN,
  T_RPAREN,
  T_LBRACE,
  T_RBRACE,
  T_ARROW,
  T_DOUB_ARROW,
  T_EQUAL,
  T_COLON,
  T_BAR,
  T_PROD,
  T_COPROD,
  T_COMMA,
  T_ARBITRARY,
  T_COMPOSE,
  T_TERM_MAP,
  T_ONE,
  T_EOF
} tokenKind;

typedef enum {
  C_BOGUS,
  C_DUMP,
  C_EDIT,
  C_HELP,
  C_INFO,
```

```
  C_QUIT,
  C_RESET,
  C_QUERY
} commandKind;

typedef struct {
  tokenKind   kind;
  boolean     error;
  char        *data;
  commandKind cKind;
} token;

extern void    ConstructLex (void);
extern void    DestructLex  (void);
extern void    ResetLex     (void);
extern boolean PreLex       (char *fromFile, boolean lazy);
extern void    PostLex      (void);
extern void    Scan         (token *nextToken);
extern char    *InsertName  (char *word);
extern int     Line         (void);
extern boolean ForcedQuit   (void);
extern boolean IsAdhoc      (void);

#endif
```

F.6 `stab.h`

This file describes the interface to the symbol table: code module `stab.c`.

```
#ifndef __STAB__
#define __STAB__

#include "chirp.h"
#include "charm.h"
#include "unify.h"

typedef enum {
  CL_VARIABLE,
  CL_FUNCTION,
  CL_TYPE,
  CL_ARBITRARY,
  CL_TYPE_REP
} classKind;

typedef enum {
  F_MACRO,
  F_COMPOSITE,
  F_CONSTRUCTOR,
  F_DESTRUCTOR,
  F_PRIMITIVE,
  F_INDUCTIVE,
  F_COINDUCTIVE
} functionKind;

typedef struct _stabEntry {
  classKind class;
  char      *name;

  int level;

  boolean fromFile;
  boolean probational;

  struct _stabEntry *next;

  union SEAttrs {
    struct {
      functionKind kind;
      int          numMacros;

      union {
        int posn;

        struct {
          COMB_CLASS       class;
          combKind         kind;
          CTERM            code;
```

```
          struct _codeType **macroSigs;
          struct _codeType *signature;
        } composite;

        struct {
          COMB_CLASS       class;
          combKind         kind;
          int              posn;
          char             *typeName;
          substitution     *strucTyping;
          struct _codeType *signature;
        } structor;

        struct {
          COMB_CLASS       class;
          combKind         kind;
          struct _codeType **macroSigs;
          struct _codeType *signature;
        } primitive;

        struct {
          COMB_CLASS       class;
          combKind         kind;
          char             *typeName;
          struct _codeType **macroSigs;
          struct _codeType *signature;
        } ductive;
      } data;
    } func;

    struct {
      boolean inductive;
      int     numStructors;
      int     numArbitrary;
      char    **structorsFor;
    } type;

    int posn;
  } attr;
} stabEntry;

extern void      ConstructStab    (void);
extern void      DestructStab     (void);
extern void      ResetStab        (void);
extern void      PostStab         (void);
extern void      PurgeStab        (void);
extern void      PushScope        (void);
extern void      PopScope         (void);
extern stabEntry *LookUpPrimitive (boolean bottom, char *name);
extern stabEntry *FindPrimitive   (boolean bottom, char *name);
extern stabEntry *InsertPrimitive (boolean bottom, char *name, classKind class,
                                   union SEAttrs *attrs);
extern void      DumpTable        (void);
extern void      PrintSub         (substitution *s);

#define LookUp(name)                       LookUpPrimitive(false, name)
#define LookUpAtBottom(name)               LookUpPrimitive(true, name)
#define Find(name)                         FindPrimitive(false, name)
#define FindAtBottom(name)                 FindPrimitive(true, name)
#define Insert(name, class, attrs)         InsertPrimitive(false, name, class, attrs)
#define InsertAtBottom(name, class, attrs) InsertPrimitive(true, name, class, attrs)

#endif
```

F.7 `trans.h`

This file describes the interface to the translator: code module `trans.c`.

```
#ifndef __TRANS__
#define __TRANS__

#include "stab.h"
#include "charm.h"

typedef enum {
  B_EMPTY,
  B_VARIABLE,
```

```
  B_PAIR
} baseKind;

typedef struct _varBase {
  baseKind kind;

  union {
    char *x;

    struct {
      struct _varBase *v0;
      struct _varBase *v1;
    } pair;
  } data;
} varBase;

typedef enum {
  R_EMPTY = 0,
  R_VARIABLE,
  R_ABSTRACTION,
  R_FUNCTION,
  R_PAIR,
  R_CASE,
  R_FOLD,
  R_MAP,
  R_UNFOLD,
  R_RECORD
} termKind;

typedef struct {
  varBase      *v;
  struct _term *t;
} CFMAEl;

typedef struct _term {
  termKind kind;

  union {
    char *x;

    struct {
      varBase      *w;
      struct _term *t_;
      struct _term *t;
    } abstraction;

    struct {
      stabEntry    *func;
      CFMAEl       *abstr;
      struct _term *t;
    } function;

    struct {
      struct _term *t0;
      struct _term *t1;
    } pair;

    struct {
      char         *type;
      CFMAEl       *abstr;
      struct _term *t;
    } caseOrFoldOrMap;

    struct {
      char         *type;
      struct _term **abstr;
      varBase      *w;
      struct _term *t;
    } unfold;

    struct {
      char         *type;
      struct _term **abstr;
    } record;
  } data;
} term;

extern CTERM Translate        (varBase *vb, term *t);
extern void  DestructVBase    (varBase *vb);
extern void  DestructTerms    (term *t);
extern void  DestructCombs    (CTERM c);
extern void  PrintCode        (CTERM code);
extern void  PrintCodeAsLogic (CTERM code);

#endif
```

F.8 unify.h

This file describes the interface to the typechecker: code module `unify.c`.

```
#ifndef __UNIFY__
#define __UNIFY__

#include "stab.h"
#include "charm.h"

typedef enum {
  TK_CONCRETE,
  TK_ARBITRARY
} typeKind;

typedef struct _codeType {
  typeKind kind;

  union {
    struct {
      struct _stabEntry *type;
      struct _codeType  **params;
    } concrete;

    int which;
  } data;
} codeType;

typedef struct {
  boolean function;

  union {
    struct _stabEntry *func;
    CTERM             code;
  } data;
} check;

extern codeType *TypeCheck   (check toCheck);
extern codeType *CopyType    (codeType *type);
extern void     DestructType (codeType *type);
extern void     PrintType    (codeType *type);

#endif
```

F.9 hash.h

This file is the header for `hash.c`. This "module" contains the code for performing the hashing which is required throughout the rest of the system.

This code is set aside by itself in the anticipation that later, other general purpose routines might need to be added to the system, but which do not logically correspond to any other particular module.

```
#ifndef __HASH__
#define __HASH__

extern int Hash (char *str, int tableSize);

#endif
```

F.10 `charm.h`

This file describes the interface between Chirp and Charm.

```
#ifndef __CHARM__
#define __CHARM__

#define CHARM_VER "v0.1 beta"

typedef enum {
  Comp,
  Comb,
  Parm,
  Unpr              /* unused */
} CTERM_TYPE;

typedef enum {
  Active,
  Ractive,
  Inactive,
  Dist,
  Cdefs
} COMB_CLASS;

typedef enum {
  CK_CASE,
  CK_FOLD,
  CK_UNFOLD,
  CK_RECORD,
  CK_MAP_L,
  CK_MAP_R,
  CK_OTHER
} combKind;

typedef struct Cterm_node *CTERM;

typedef struct Cterm_list_node *CTERM_LIST;

typedef struct Cterm_list_node {
  CTERM      cterm;
  CTERM_LIST next;
} CTERM_LIST_REC;

typedef struct Cterm_node {
  CTERM_TYPE tag;

  union {
    struct {
      CTERM cterm1;
      CTERM cterm2;
    } composition;

    struct {
      char       *name;
      COMB_CLASS class;
      combKind   kind;
      CTERM_LIST cterm_list;
    } combinator;

    int position;
  } type;
} COMB_TERM;

typedef enum {
  ST_NONREC,
  ST_REC
} subType;

typedef enum {
  F_1,
  F_ID,
  F_PROD,
  F_OTHER
} factorizer;

typedef struct _substitution {
  factorizer kind;

  union {
    struct {
      subType arg;
      int     posn;
```

```
      char    *type;
    } id;

    struct {
      struct _substitution *s1;
      struct _substitution *s2;
    } prod;

    struct {
      char                *type;
      int                 numSubs;
      struct _substitution **s;
    } other;
  } data;
} substitution;

typedef enum {
  S_FAILURE,
  S_SUCCESS,
  S_REPLACE
} status;

extern substitution *SubLookup   (char *structor);
extern int          StructorPosn (char *structor);
extern char         **CombsFor   (char *type);

extern status Compile (char *name, CTERM code);
extern CTERM  Machine (char *name);

#endif
```

Appendix G

Source Code for Chirp

As of April 2, 1993, Chirp is comprised of almost 7000 lines of Ansi C source code (not including documentation), which compiles under the BSD Unix operating system (tested under SunOS 4.1.2) with the Gnu C Compiler (gcc).

The files in table G.1 make up the entire Chirp system[1], although the list is subject to change. The source is available upon request (see also appendix F). A bundled package including Chirp/Charm, other related files, relevant papers, and examples should also be obtainable soon. On University of Calgary Computer Science machines, an executable system is available in `~marc/charity`.

[1] Note that `help` and `info` are not source code, but are used by the system when the user invokes the *help* or *info* screens.

chirp.h	chirp.c
parse.h	parse.c
error.h	error.c
lex.h	lex.c
stab.h	stab.c
trans.h	trans.c
unify.h	unify.c
charm.h	interface.c
hash.h	defaults.h
help	info

Table G.1: Ansi C source files for Chirp.