

App

An Algebraic Typing and Pattern Matching Preprocessor for C++

Table of Contents

- [PREFACE](#)
- [Introduction](#)
- [Getting Started](#)
 - ◆ [Installation](#)
 - ◆ [Usage](#)
 - ◆ [Modes of Operation](#)
 - ◆ [Sample Code](#)
 - ◆ [Caveats](#)
- [Alpha Type Declarations](#)
 - ◆ [Alpha Base Types](#)
 - ◇ [user defined body or handle classes](#)
 - ◇ [shorthand notation for implied user defined body and handle classes](#)
 - ◇ [generated extendible body and handle classes](#)
 - ◇ [user defined body and generated extendible handle classes](#)
 - ◇ [explicit keyword](#)
 - ◇ [last base product component](#)
 - ◇ [overriding generated write function via user defined print function](#)
 - ◆ [General Algebraic Types](#)
 - ◇ [alpha type translation](#)
 - ◇ [handle and body class hierarchies](#)
 - ◆ [Common Handle Class Methods](#)
 - ◇ [static nil function and null objects](#)
 - ◆ [Alpha Data Types](#)
 - ◇ [zero as null and zero sums](#)
 - ◇ [singleton sets](#)
 - ◇ [mutability and custom access](#)
 - ◇ [ctor parameter passing mechanisms](#)
 - ◇ [explicit keyword](#)
 - ◇ [root classes and member function definition restrictions](#)
 - ◇ [alpha-type-spec construct](#)
 - ◆ [Alpha Unit Types](#)
 - ◇ [explicit keyword](#)
 - ◆ [Alpha Type Types](#)
 - ◆ [Alpha Forward Types](#)
 - ◇ [unit types are nonrecursive](#)
 - ◇ [properly nested forward declarations](#)
 - ◇ [forward root classes and user code](#)
 - ◆ [Special Considerations](#)
 - ◇ [the constant zero](#)
 - ◇ [null construction](#)
 - ◇ [ctors as conversion operators](#)
 - ◇ [unary ctors](#)
 - ◇ [copy ctors as identity operators](#)

- ◊ [static constructor functions](#)
- [Match Statements](#)
 - ◆ [General Form of Match Statements](#)
 - ◆ [Subject Expressions](#)
 - ◆ [Pattern Matching Translations](#)
 - ◊ [pattern classifications](#)
 - ◊ [general translation scheme](#)
 - ◆ [Inductive Patterns](#)
 - ◊ [alpha type type references](#)
 - ◊ [alpha base type references](#)
 - ◆ [Binding Patterns](#)
 - ◊ [wildcards](#)
 - ◊ [uniqueness of binding patterns](#)
 - ◆ [Zero Patterns](#)
 - ◆ [Parametric Pattern Types](#)
 - ◊ [parametric base types](#)
 - ◆ [Nonpositional Patterns](#)
 - ◆ [Runtime Type Selection](#)
 - ◊ [option stor method](#)
 - ◆ [Case Coverage](#)
 - ◊ [default cases and short-circuiting](#)
 - ◊ [safety cases](#)
- [The Runtime Library](#)
 - ◆ [Alpha Component](#)
 - ◆ [Beta Component](#)
 - ◊ [alpha type declarations](#)
 - ◊ [List member functions](#)
 - ◊ [List template functions](#)
 - ◊ [Single type](#)
 - ◊ [Associative List class](#)
 - ◊ [Extending List class](#)
 - ◆ [Additional Components](#)
 - ◊ [Str class](#)
 - ◊ [Pretty Printer class](#)
 - ◊ [Flow class](#)
 - ◊ [Shared pointer classes](#)
 - ◆ [Symbol List](#)
- [Hacker's Guide to the Source](#)
 - ◆ [Physical Structure](#)
 - ◆ [Logical Structure](#)
 - ◆ [Customizing](#)
 - ◆ [Special Considerations](#)
 - ◆ [Projects](#)
- [The App Input Grammar](#)
- [Glossary](#)
- [Changes](#)

This is the APP and APPLIB version 2.2 documentation.

Copyright (C) 1999, 2000 George Nelan.

Permission is granted to make and distribute verbatim copies of this documentation provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this documentation under the conditions for verbatim copying, provided also that the files [COPYING.txt](#) and [LCOPYING.txt](#) in this directory are included unchanged, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this documentation into another language, under the above conditions for modified versions, except that the files [COPYING.txt](#) and [LCOPYING.txt](#) in this directory may be included in translations approved by the Free Software Foundation instead of in the original English.

Comments, suggestions, etc. welcome.

George Nelan

Mail: georgen@primenet.com

App homepage: <http://www.primenet.com/~georgen/app.html>

[CONTENTSNEXT](#)

Introduction

'App' is a preprocessor for C++ that accepts as input arbitrary C++ code that may contain embedded constructs for specifying algebraic data types and associated pattern matching operations, and produces as output the same code with all such constructs translated to normal C++. 'Applib' is the associated runtime library that supports the core run time requirements of the translated code, and which provides additional utilities.

The notion of algebraic types comes from the world of functional programming, where such types are relied on as a way to reason about program semantics. For example in the functional language [Haskell](#) one can define an algebraic parametric type for trees with arbitrary labels as follows.

```
data Tree x = Tip x | Node x (Tree x) (Tree x)
```

This type is parametric because the type of tree labels x is a parameter. To count the number of tree tips one can define a pattern matching function.

```
ntips (Tip _) = 1
ntips (Node _ left right) = ntips left + ntips right
```

The ntips function is defined over all such trees because (by examination) it is defined over all structures of such trees. This is an example of definition by structural induction, in which simple structures (e.g. Tip) form base cases, and compound structures (e.g. Node) form inductive cases.

More pragmatically, algebraic types can be considered forms of discriminated unions. The cases of Tree (i.e. Tip and Node) can be considered injectors of discreet types and the whole type (i.e. Tree) can be considered a union of such types together with the means for discriminating at run time between the individual cases in order to project the constituents of the cases.

More formally, algebraic types are sums of products, and pattern matching is a more or less formal way of doing casing.

The casing style is certainly not limited to the functional world. It is straightforward enough to do pretty much the same thing in C++ (notwithstanding the apparent lack of formality).

```
template <class X>
class Tree {
public:
    virtual ~Tree() {}
};

template <class X>
class Tip : public Tree<X> {
public:
    X label;
    Tip(X x) : label(x) {}
};

template <class X>
class Node : public Tree<X> {
public:
    X label;
    Tree<X>* left;
    Tree<X>* right;
    Node(X x, Tree<X>* l, Tree<X>* r) : label(x), left(l), right(r) {}
};
```

```

template <class X>
int ntips(Tree<X>* t) {
    if (dynamic_cast<Tip<X>*>(t))
        return 1;
    else if (Node<X>* u = dynamic_cast<Node<X>*>(t))
        return ntips(u->left) + ntips(u->right);
    else
        return 0;
}

void test_0() {
    Tree<int>* t = new Node<int>(1, new Tip<int>(2), new Tip<int>(3));
    cout << ntips(t) << endl;
}

```

Naturally there are some linguistic differences such as the use of pointers and explicit dynamic casting & allocation, but the basic idea is the same. However the casing style as illustrated above can itself be a source of problems especially for less trivial examples: What if a dynamic cast isn't done just so? What if the underlying case structure changes, and how does one know if all such changes are accounted for?

In a functional language like Haskell, there are no `dynamic_casts` or the like, at least not at the linguistic level. And if the underlying case structure changes, it is technically not difficult for a functional language compiler to let the programmer know about it, because all relevant cases are statically related by the algebraic typing system. All the compiler really has to do is a simple case analysis of related patterns, and check that all relevant cases are at least mentioned, as pretty much all that is needed is to make sure the programmer is at least aware of changes to the underlying structure as defined by the type equations.

It would appear then that at least theoretically, functional languages such as Haskell simply do not suffer from the difficulties that the casing style tends to impose on most other kinds of languages including C++. This is where `app` comes in. What `app` essentially does is provide for C++ pretty much the same capabilities that functional languages have regarding algebraic types. In a sense `app` can be viewed as bringing C++ closer to the functional programming world, although perhaps just as well `app` can be viewed as directly supporting certain kinds of visitor-style patterns in C++, insofar as algebraic types can be viewed as supporting such patterns. In any event, here is the tree example, done the `app` way.

```

namespace alpha
{

$data Tree<class X>
= Tip(X label)
| Node(X label, Tree<X> left, Tree<X> right)
;

} // namespace alpha

using namespace alpha;

template <class X>
int ntips(Tree<X> t) {
    int result;
    $match (t) {
        (Tip<X>(_)) => {result = 1;}
        (Node<X>(_, left, right)) => {result = ntips(left) + ntips(right);}
    }
    return result;
}

void test_1() {
    Tree<int> t = Node<int>(1, Tip<int>(2), Tip<int>(3));
}

```

```

    cout << ntips(t) << endl;
}

```

The output from app for this example follows verbatim (option `--pretty`). Notice that the generated code does not use pointers directly. The handle/body idiom with reference counting is employed instead; class 'Top' forms the base class of the handle hierarchy, and class 'Top__' the base class of the body hierarchy. It should also be noted that in app terminology "base" is typically used when referring to relatively simple kinds of algebraic types. Context should suffice to distinguish between the senses.

```

namespace alpha
{

template<class X>
class Tree : public Top {
public:
    Tree() : Top() {}
    Tree(Top__* handle, Copy docopy = nocopy) : Top(handle, docopy) {}
};

template<class X>
class Tip__ : public Top__ {
    X label_;
public:
    const char* raw_type() const {return "Tip";}
    Tip__(X label) :
        label_(label)
    {}
    X label() {return label_;}
    void write(ostream& _os_) {
        show_begin(_os_, form_using_rtti() ? typeid(*this).name() : "Tip");
        _os_ << label();
        show_end(_os_);
    }
};

template<class X>
class Tip : public Tree<X> {
public:
    Tip(X label) :
        Tree<X>(new Tip__<X>(label))
    {}
    Tip(Top__* handle, Copy docopy = copyok) :
        Tree<X>(handle, docopy)
    {}
    Tip__<X>* operator->() const {
        return static_cast<Tip__<X>*>(handle);
    }
    Tip__<X>& operator*() const {
        return static_cast<Tip__<X>&>(*handle);
    }
    static Tip<X> nil() {return Tip<X>(0, nocopy);}
};

template<class X>
class Node__ : public Top__ {
    X label_;
    Tree<X> left_;
    Tree<X> right_;
public:
    const char* raw_type() const {return "Node";}
    Node__(X label, Tree<X> left, Tree<X> right) :

```

```

    label_(label),
    left_(left),
    right_(right)
}
X label() {return label_;}
Tree<X> left() {return left_;}
Tree<X> right() {return right_;}
void write(ostream& _os_) {
    show_begin(_os_, form_using_rtti() ? typeid(*this).name() : "Node");
    _os_ << label();
    show_delim(_os_);
    _os_ << left();
    show_delim(_os_);
    _os_ << right();
    show_end(_os_);
}
};

```

```

template<class X>
class Node : public Tree<X> {
public:
    Node(X label, Tree<X> left, Tree<X> right) :
        Tree<X>(new Node__<X>(label, left, right))
    {}
    Node(Top__* handle, Copy docopy = copyok) :
        Tree<X>(handle, docopy)
    {}
    Node__<X>* operator->() const {
        return static_cast<Node__<X>*>(handle);
    }
    Node__<X>& operator*() const {
        return static_cast<Node__<X>&>(*handle);
    }
    static Node<X> nil() {return Node<X>(0, nocopy);}
};

```

```

} // namespace alpha

```

```

using namespace alpha;

```

```

template <class X>
int ntips(Tree<X> t) {
    int result;

    //$$ begin $match
    { bool _alpha_fail = true;
      Tree<X> _alpha_norm_1 = t;
      if (dynamic_cast<Tip__<X> *>(_alpha_norm_1())) {
          Tip<X> _alpha_cast_1(_alpha_norm_1());
          _alpha_fail = false;
          //$$ begin user statement code
          result = 1;
          //$$ end user statement code
      }
      if (_alpha_fail && dynamic_cast<Node__<X> *>(_alpha_norm_1())) {
          Node<X> _alpha_cast_1(_alpha_norm_1());
          Tree<X> left = _alpha_cast_1->left();
          Tree<X> right = _alpha_cast_1->right();
          _alpha_fail = false;
          //$$ begin user statement code
          result = ntips(left) + ntips(right);
          //$$ end user statement code
      }
    }
}

```

```
    }
    if (_alpha_fail) escape("match failed");
  }
  //$$ end $match

  return result;
}

void test_1() {
  Tree<int> t = Node<int>(1, Tip<int>(2), Tip<int>(3));
  cout << ntips(t) << endl;
}
```

[CONTENTSNEXT](#)

Getting Started

Installation

If you need to download the code, you can do so from the app homepage listed in the [PREFACE](#). At the download page you should find a file named `alphaxy.zip`, containing the APP/APPLIB version `x.y` distribution. This is a zip file that will unzip into a directory named `alpha`. You should first move aside your existing `alpha` directory if you have already downloaded a previous version. For Windows you can use `winzip`; for linux/unix use `"unzip -a alphaxy.zip"`.

This software was initially developed on a W95 Intel platform with MS VC++ 5.0, and presently runs under both VC++ and g++ 2.91.66+ (cygwin/slackware 7 linux). The `alpha` directory when unzipped is set up for both VC++ and g++ environments. See first the `README.txt` file after unzipping. This file will tell you how to install for any of: VC++, linux/unix g++, cygwin g++, and dual VC++/cygwin g++ platforms.

For MS VC++, installation is simple. About all `install.bat` does is copy the pre-built executables, static library, and header files to the `bin`, `lib`, and `include` directories of your choice – it may also invoke `app` to fix `#line`'s in `_beta.h`, depending on the `INCLUDE` variable setting.

For g++, installation is a little more involved, as there are no pre-built g++ executables or libraries. Basically everything has to be compiled first, after which installation proceeds pretty much as for VC++. This is all handled by a single script file `gboot.sh`.

Usage

`App` is a command-line driven application. Type `'app -h'` to get an idea of what the command line arguments look like, and `'app -v'` to get current version related information.

General usage is `'app options f1 f2 ... fn'`. `App` expects zero or more input files `f1 f2 .. fn`. In the zero case, input is taken from standard input, and output is by default to standard output. In the one case input is taken from `f1`, and output is by default to `_f1`. In the two or more cases, input is taken from the ordered catenation of `f1 f2 .. fn`, and output by default is to `_fn`; additionally a temporary file `fn.tmp` is generated that `app` uses to concatenate `f1 f2 ... fn` and is removed upon successful completion. Option `'t=x'` can be specified to indicate an extension `.x` rather than `.tmp`. You can in any case specify `'-o=f'` to override output defaults for filename `f` ("`-`" for `f` is taken as `cout`).

In the multiple file input case, `f1 f2 .. fn-1` are taken as implicit include files. You can use option `'-i=d'` one or more times to tell `app` where to look for any of the `f1 f2 .. fn-1` that can't be found otherwise, where `'d'` is a directory name. The precedence order for `'-i'` is right to left so that e.g. `'app -i=d1 -i=d2 ...'` implies that `d2` is searched before `d1`.

Option `'-ori'` (`'--only_read_includes'`) causes `app` to treat the argument files `f1..fn-1` as implicit include files as usual but which are excluded from the generated code. By using explicit `#include` directives for `_f1.._fn-1` in your code together with this option (assuming default `_f1.._fn-1` `app` outputs), you can reduce the size of the generated code and also the number of redundant symbols. This is particularly effective for use with class browsers etc.

`App` accounts for `#line` directives in input, but doesn't do C++ preprocessing functions itself. As it can be operated in a mode that makes it appear as an extension to the C++ preprocessor, for the sake of modal consistency it just uses a simple command line facility for include handling rather than trying to deal with

`#includes` in input or some kind of special `#include`-like keyword that only `app` would understand.

This can have implications when files included by the `app` command line facility themselves include other files; as `app` does not know about the other included files it may be necessary to compile the `app` generated result by specifying additional include directories as compiler parameters so as to account for the other files. If things are not set up properly and there are problems in a source file that the compiler wants to complain about, the compiler may get confused and refer to nonexistent or otherwise bogus files due for example to `app` generated `#line` directives that don't correspond with reality. In such cases option `'-nolds'` can be used to get `app` to not generate `#line` directives; the compiler will then at least be able to refer to an existing file. Option `'-lfp=s'` can also be used to get `app` to attach `s` as a filename prefix to all generated `#line` directives; this may prove useful in certain circumstances.

By default `app` deliberately generates "unpretty" code by removing redundant whitespace. An option `'-p'` (`'--pretty'`) can be specified to (largely) preserve original whitespace, and not generate `#line` directives. The pretty option can be useful for certain illustrative purposes but can cause headaches because the pretty code looks almost exactly like the original, and even the synthesized code looks pretty, so it is easy to edit it by mistake instead of the original. Beware.

Modes of Operation

There are at least three ways that `app` can operate. First, it can be utilized as a standalone program, which is surely the easiest way, and it isn't that painful to deal with either. Second, it can be utilized as a frontend for the C++ preprocessor. Third, it can be utilized as a backend for the C++ preprocessor.

The standalone mode works pretty much as expected. You just run `app` like any other code generating program such as `lex` or `yacc`, then arrange to compile the output. If you have makefile capabilities it's just a one or two step process. I use the Cygnus `cygwin` free software, which provides for certain Unix-like utilities for Windows including `'make'`, so I just type `'make'` in a shell, which runs `app` as needed, then switch to Visual Studio and hit F7, and that's about all there is to it.

By default `app` generates a `'#define APP'` directive as the first line of output. This is useful for IDEs that are sensitive to filename extensions. For example I like to use `.cpp` files for both `app` inputs and outputs because the MS Visual Studio editor highlights C/C++ keywords – but only for C/C++ files. By wrapping all `app` input code with `#ifndef APP ... #endif` directives the VC++ compiler won't complain about `app` constructs even though the `app` inputs are part of the same project as the outputs, because only `app` outputs get the requisite `#define`. Option `'-noad'` can be used to get `app` to not generate the `#define`, which might come in handy when `app` is utilized as a backend for the C++ preprocessor – when it's too late for `#define`'s.

`App` can also operate as a C++ preprocessor frontend, that is, if you can get the C++ preprocessor to run independently of the compiler, or if the compiler can act like an independent preprocessor in case it also happens to implement the preprocessor. In this case though you probably want to use a makefile or some such to get things to work out. However if you can do all that it would probably be better to use `app` as a backend for the C++ preprocessor in order to better take advantage of its capabilities. Don't forget the `'-noad'` option.

Sample Code

The sample directory contains an assortment of simple examples illustrating the basic concepts of `app` and `applib`. There is just one program `sample_main` that links with the individual samples via `sample_x()` interface routines. Many of the `sample_x.cpp` files have a corresponding `_sample_x.cpp` file, the `app` generated version.

If you are using MS VC++ the sample directory is also a project, and you should be able to run the project in

the usual manner. Note that as object files are not included in the distribution, when you run the project you will be told that there are missing files and asked if you would like to rebuild them; it doesn't matter how you answer.

If you are using g++ the gsample directory also contains a very simple hello world program (app-style) and associated makefile that you may wish to examine first.

You may also wish to examine the app source as substantial portions of it are written in itself. As for the sample code, each app input file f.h or f.cpp has a corresponding output file _f.h or _f.cpp. The various app makefiles also illustrate pretty well how to deal with moderately complex app related build environments using some of the more interesting GNU make features.

Caveats

App uses flex to do input scanning, but the code is not particularly flex specific. Unfortunately flex appears to have a problem (at least on Windows) when the last line of input is not newline terminated. You should make sure all app input files have properly terminated last lines otherwise final input tokens may silently disappear.

As of g++ 2.95.2, there are certain limitations under g++: option '-ori' is ineffectual, and #line directives that point back to the generated output file are not generated (the app code specializes the filebuf and ostream classes and that appears to be a problem with g++ due to buggy iostream support).

[CONTENTSNEXT](#)

Alpha Type Declarations

App accepts as input arbitrary C++ code that may contain embedded constructs for specifying algebraic data types and associated pattern matching operations. The constructs for specifying algebraic types are alpha type declarations, and algebraic types are said to be alpha types herein. The term 'alpha' is taken as a kind of app specific synonym for 'algebraic'.

When app encounters an alpha type declaration it is subjected to a translation process that generally produces corresponding C++ class declarations. App is no smarter than it needs to be however which for one thing means that it just assumes that whenever such a declaration is encountered, it appears in a context valid for C++ class declarations. If you specify such a declaration in some other context, say in the middle of an expression, app knows nothing of it and will happily generate class declarations in the middle of the expression anyway.

App generates class declarations largely formatted according to a number of skeletons, a kind of macro definition facility, and which can be user-defined, both at app build time and run time (`--skeletons=f` option), providing for a significant degree of customization in the generated code; see the file `skeletons.txt` in the `skel` directory, and also the source for additional information. For most purposes the default skeletons should suffice.

There are four kinds of alpha types, base types, unit types, data types, and type types, and correspondingly, four kinds of alpha type declarations. There are also forward type declarations for those types that may be defined mutually recursive.

Alpha Base Types

Alpha base types form the base case for the alpha typing system as it were. Non-base alpha types are defined in terms of other alpha types. Alpha base types provide the means for introducing arbitrary C++ types into the system. The app input grammar portion for alpha base type declarations follows. For the complete grammar and related notation see "[The App Input Grammar](#)".

```
alpha-base-type-decl
-> '$base' 'explicit'? id alpha-type-head? base-products code? ';'
-> '$base'          id alpha-type-head? ';'

base-products
-> '(' balanced-items-list? ')

user
-> '{' balanced-item* '}'

user-code
-> '$user'? user

body-user-code
-> user-code

handle-user-code
-> user-code

code
-> body-user-code handle-user-code?

alpha-type-head
-> '<' alpha-type-head-param/',' ' >'
```

```
alpha-type-head-param
-> 'class' id
```

App does not attempt to parse arbitrary C++ code. However when arbitrary C++ code can appear embedded within app specific constructs, app must still figure out which is which. App accomplishes this by assuming that such C++ code is "balanced" with respect to parenthesis, braces, and brackets. This same assumption also allows app to sort out comma separated lists of arbitrary C++ code. If said C++ code is syntactically incorrect with respect to balancing, app will notice and complain. (The various balanced-xxx grammar rules relate to this syntactic balancing act).

Here is an example of an alpha base type declaration that includes most of the optional syntactic constructs.

```
$base Foo<class A>(A x, int y) $user {} $user {};
```

This states that "Foo" is a parametric base type with two constituent base products, A x and int y. The presence of "\$user {} \$user {}" says two things; first, that the body class Foo__ is user defined, corresponding to the first "\$user {}" construct, and second, that the handle class Foo is also user defined, corresponding to the second construct. Both classes are template classes (parametric alpha types correspond to template classes). Note that "user defined" really just means "not app defined".

This declaration does not have corresponding generated classes. Such a situation may arise when the skeletal code for base classes insufficiently describes the desired generation format. App still needs to know that Foo is an alpha type however and so in such situations this method can be utilized. Another potential use for this method is when an independent C++ type already exists and it is desired to make it appear as an alpha type. The standard C++ string class can be made to appear as an alpha type in this way for example.

```
$base string() $user {} $user {};
```

The second alpha-base-type-decl grammar rule allows shorthand notation for when base-products is "()" and code is "\$user {} \$user {}":

```
alpha-base-type-decl
-> '$base' 'explicit'? id alpha-type-head? base-products code? ';'
-> '$base' id alpha-type-head? ';'

```

The string declaration above for example can also be stated as follows.

```
$base string;
```

Any named C++ type can be made to appear as an alpha base type in the same manner. As "\$user {} \$user {}" effectively states that the user ("not app") has full responsibility for the implementation, it is not necessary that such types adhere to any particular implementation idiom, including the handle/body idiom app employs (thus C++ native types can also appear as alpha base types). They may need to support stream output however (via operator <<), depending on usage particulars – alpha types in general support stream output. Whether a given C++ type should or needs to appear as an alpha type depends on if it is referenced from a context in which alpha typing is assumed, e.g. from another alpha type, or from a match statement (as part of a pattern).

Specifying "\$user {...}" as a user-code construct means that the associated body (or handle) class is user defined and that whatever "..." is in the accompanying "{...}" is emitted as-is. When "\$user" is specified usually there is no "..." as the definition is typically placed elsewhere. (This applies to all alpha types for which "\$user" can be specified – not just base types).

The user-code construct allows for "{...}" instead of "\$user {...}". This means that the corresponding body (or handle) class is generated, but also extended according to "...", taken as a replacement for the skeletal

variable named `$user_part$`. (Skeletons generally have one or more variables `X` that are replaced at generation time according to syntactic particulars).

Another way to specify a code construct is `"$user {}"` or `"$user {} {...}"` which says that, while the corresponding body class is user defined, the handle class is app generated, and possibly extended according to "...". The Atom alpha type in the beta.h library file is an example of a base type declaration with a `"$user {} {...}"` code construct.

```
class Atom__ : public Top__ {
    char* data_;
public:
    Atom__(const char* data) : data_(new_string(data)) {}
    Atom__(const char* data, bool clone) :
        data_(clone ? new_string(data) // safe but maybe less efficient
                : const_cast<char*>(data)) // efficient but maybe less safe
    {}
    ~Atom__() {delete[] data_;} // *assumes* safe ownership of new[]'d buffer
    const char* data() {return data_;}
    bool print(ostream& os) {write_string(os, data()); return true;}
};

$base Atom(const char* data) $user {} {
    Atom(const char* data, bool clone) :
        Top(new Atom__(data, clone), nocopy)
    {}
};
```

Here there is just one base product `"const char* data"`. If you look at the default skeletons, you will notice that there are no explicit provisions for handling anything other than "simple" base products regarding ctor member initializations, where a "simple" base product is something that involves a native type like `"int x"` or, in general, something that behaves like a native type. The dynamic allocation requirements for the `data_` member above precludes use of the default skeleton in this case. Note also that the additional body ctor (with the 'clone' argument) requires a corresponding handle ctor.

The first alpha-base-type-decl grammar rule allows for optional specification of the 'explicit' keyword:

```
alpha-base-type-decl
-> '$base' 'explicit'? id alpha-type-head? base-products code? ';'
-> '$base'          id alpha-type-head? ';' ;
```

The keyword has the effect of prefixing the generated handle class ctor that instantiates the associated body object with the C++ 'explicit' keyword. This can be useful for certain kinds of unary ctors. The keyword does not apply to the second grammar rule because in that case there are no generated classes.

The last component of each base product (a balanced-item-no-comma construct) is required to be an identifier `X` that implicitly names a corresponding member variable `X_` (the preceding components presumably denote the type of `X_`). The default skeletons are defined such that all member variables corresponding to products are so named and private, and that access to such members is always by means of a public access function `X()`. For example given `"const char* data"` as a base product, the member variable is `"data_"`, and `"data()"` is the access function. Access functions for base types are automatically generated unless `"$user"` is specified for the body class.

The `Atom__` body class also defines a bool function 'print' that returns true after printing. In the alpha.h library file, the `Top__` base class for the body hierarchy defines a non-virtual wrapper function 'show' that coordinates output by first calling a protected virtual function 'print' that by default returns false, and then (if print returns false) calling protected virtual function 'write' that by default just writes '?'; the write function is

app generated for unit and data types but not base types. By defining 'print' accordingly in a base body class, you can arrange for that alpha type to generate output as desired (this holds for any alpha type – not just base types). The stream output operator:

```
ostream& operator << (ostream& os, Top& t)
```

is defined in alpha.h so as to apply to all alpha types. See also the comments in beta.h about formatting conventions.

The handle class generated for the Atom base type declaration follows.

```
class Atom : public Top {
public:
  Atom() {}
  Atom(const char* data) :
    Top(new Atom__(data), nocopy)
  {}
  Atom(Top__* handle, Copy docopy = copyok) : Top(handle, docopy) {}
  Atom__* operator->() const {
    return static_cast<Atom__*>(handle);
  }
  Atom__& operator*() const {
    return static_cast<Atom__&>(*handle);
  }
  static Atom nil() {return Atom(0, nocopy);}

  Atom(const char* data, bool clone) :
    Top(new Atom__(data, clone), nocopy)
  {}
};
```

The alpha base type mechanism allows for the introduction of arbitrary C++ types either by means of suitable specifications for base products or by using the \$user construct. However, as the last component of each base product is required to be an identifier corresponding to a member variable, in order to introduce certain C++ types in this way, such as arrays or function pointers, you need to do so indirectly, using typedefs or the like. Alternatively, you can define base types as you wish via \$user, and if necessary, use accessor functions instead of pattern matching (pattern matching with base types is limited and involves certain assumptions).

General Algebraic Types

A general nonparametric algebraic type T has the abstract form:

$$T = S_1T + \dots + S_mT$$

where $S_1T \dots S_mT$ are $m \geq 1$ sum components such that for $1 \leq i \leq m$ each component S_iT has the abstract form:

$$I_iT (P_iT_1 * \dots * P_iT_n)$$

where I_iT is the S_iT injector (or itor), and $P_iT_1 \dots P_iT_n$ are $n \geq 0$ product components of S_iT such that for $1 \leq j \leq n$ each product P_iT_j mentions an algebraic type, and where n is a particular function of i .

Parametric algebraic types introduce type parameters such that any of the P_iT_j may depend on said parameters in a manner analogous to the way members of a C++ template class may depend on template parameters. Indeed alpha parametric types are defined in terms of C++ template classes with template parameters serving as alpha type parameters in the obvious way.

App translates an alpha type T corresponding to a general nonparametric algebraic type T abstractly as

follows.

```
class T      : public Top {...};
class IiT__ : public Top__ {P1T1' ... P1Tn' ...};
...
class ImT__ : public Top__ {PmT1' ... PmTn' ...};
class IiT   : public T {... IiT__* operator->() {return ... (handle);} ...};
...
class ImT   : public T {... ImT__* operator->() {return ... (handle);} ...};
```

Here each PiT_j ' denotes a product translation, which in brief amounts to an appropriate C++ member variable declaration. All such declarations are private; there are corresponding public access functions automatically generated by default, but which may be user defined. The various public $\text{operator->}()$'s provide access to body classes $IiT_$ from handle classes IiT . For parametric types, template declarations and parameters are included in the translation in the obvious way, noting that neither Top nor $Top_$ are parameterized.

Thus in general the handle class hierarchy is three deep, with Top at the top level, classes T (also called root classes) at the middle level, and classes IiT at the bottom level; and the body class hierarchy is two deep, with $Top_$ at the top level, and classes $IiT_$ at the bottom level. The hierarchies are usage-connected by means of the various smart pointer operators, in effect forming a kind of bridge pattern.

Conspicuously absent in the translation is the equivalent of a $T_$ class. The essential reason for this is that such classes, while technically feasible, would be redundant if included. For dynamic purposes, the handle hierarchy primarily provides an interface that includes bridging services to the body hierarchy, which primarily provides implementations that manage object memory requirements and so forth; but an intermediate $T_$ class would have no real useful role in the hierarchy as the Top & $Top_$ base classes provide common services. For static purposes, the T classes essentially just enforce the algebraic typing constraints, which if also done by $T_$ classes would again be feasible but needless.

It should probably be mentioned however that there is no real reason to suppose that this is an absolute state of affairs; if it should turn out that such classes might be useful after all, then it would not be all that difficult to get app to accomodate.

Common Handle Class Methods

The default skeletons define a set of functions that generated handle classes have (excepting root classes) including $\text{operator->}()$ and $\text{operator}^*()$, thereby allowing access to body functions from handle objects via the usual smart pointer operations, e.g.

```
Atom x = "hello world";
cout << "The length of " << x << " is " << strlen(x->data()) << endl;
```

A static function $\text{nil}()$ is also provided that returns a null alpha object of the appropriate type – a null alpha object is a handle class object with a null handle (i.e. one that references no body object). This is equivalent to handle class default construction for classes with non-nullary ctors (those having one or more arguments such as $Atom$), but the function is convenient for classes that do have nullary ctors (having no arguments) because in such cases the default handle class ctor instantiates a new body class object.

Generated handle classes have default ctors according to the default skeletons, but only those with non-nullary ctors generate null objects as a result of default construction. For example the handle class generated for the base type declaration

```
$base Empty();
```

has a nullary ctor, and default construction does not generate a null object:

```

class Empty : public Top {
public:
    Empty() : Top(new Empty__(), nocopy) {}
    Empty(Top__* handle, Copy docopy = copyok) : Top(handle, docopy) {}
    Empty__* operator->() const {
        return static_cast<Empty__*>(handle);
    }
    Empty__& operator*() const {
        return static_cast<Empty__&>(*handle);
    }
    static Empty nil() {return Empty(0, nocopy);}
};

```

Alpha Data Types

Alpha data types are the app equivalent of general algebraic types. The grammar portion for alpha data type declarations follows.

```

alpha-data-type-decl
-> '$data' id alpha-type-head? '=' sums root-code? ';'

sums
-> sum/'|'

sum
-> 'explicit'? id products code?
-> '0'

products
-> '(' product-list? ')'

product-list
-> product/','

product
-> product-decl accessor

product-decl
-> 'const'? alpha-type-spec '&'? id

accessor
-> '=>' '(' ')'
-> '=>' 'const' '(' ')'
-> '=>' 'const'
->

root-code
-> ':' handle-user-code

alpha-type-spec
-> id alpha-type-params?

alpha-type-params
-> '<' alpha-type-param/',' '>'

alpha-type-param
-> id alpha-type-params?

```

Here is an example of an alpha data type declaration together with a couple of function definitions taken (incompletely) from the beta.h library code.

```

$data List<class A> = 0 | Cons(A a, List<A> b) {
  A a(A new_a) {return a_ = new_a;}
  List<A> b(List<A> new_b) {return b_ = new_b;}
  bool print(ostream& os);
} : {
  A head() const;
  List<A> tail() const;
  int length() const;
};

template <class A>
bool Cons__<A>::print(ostream& os) {
  show_begin(os, form_using_rtti() ? typeid(*this).name() :
             form_using_zero() ? 0 : "Cons", "[");

  os << a();
  for (List<A> xs = b(); xs(); xs = xs.tail()) {
    show_delim(os);
    os << xs.head();
  }
  show_end(os, "]");
  return true;
}

template <class A>
int List<A>::length() const {
  int result = 0;
  for (List<A> xs = *this;;) {
    $match (xs) {
      (Cons<A>(_, xs1)) => {
        result++;
        xs = xs1;
      }
      (0) => {break;}
    }
  }
  return result;
}

```

Any alpha object can be null as for example when default constructed or when resulting from `nil()` function application. The specification of '0' as a sum component in an alpha data type declaration conveys the notion that zero (null) is a nominal or expected value for at least some objects of a given alpha type.

Looking at this from an implementation point of view, the handle/body idiom does not necessarily remove pointers, it just abstracts them. As pointers in general can be null so too can alpha objects, and so must be accounted for. As at least some pointers can be null as nominal values, the grammar provides for the explication of such values as sum components, and that is why 0 is a sum component for `List<A>` for example.

There are also general means for null testing: `Top__ * Top::operator ()`, as illustrated by the expression "`xs()`" in the for-loop of the print function above, `bool Top::null()`, and `bool Top::not_null()`.

Some alpha types are naturally associated with singleton sets, in that they may have one value that can associate with the empty set, and other values that can associate with a non-empty singleton set of such types. In such cases it may be tempting to use null to represent associated empty set values, and the value itself to represent the associated singleton set element. Using null in this way can be error-prone however and is also insufficient if null itself can be an element, as there is then no way to distinguish between the empty set and a non-empty set with a single null element.

One way to deal with such cases is to use an alpha data type with a '0' sum, although that can be inconvenient

if done just for that purpose. The beta.h library code provides a parametric type `Single` that may be generally used for such purposes; it models singleton sets using a '0' sum for the empty case.

```
$data Single<class A> = 0 | Single_Element(A a) : {
  bool empty() const {return null();} // true if zero constructed
  bool not_empty() const {return not_null();} // false if zero constructed
  A element() const; // a if Single_Element(A a) constructed
};
```

Note that `Single<Top>` is valid, and `Single_Element<Top>(Top())` is a singleton set with a null element resulting from application of `Top()`. This in effect provides an alternate way to represent boolean values, with a non-empty set akin to "true", and the empty set akin to "false".

Rather than using a '0' sum, it would be possible to define `List<A>` in the example above more traditionally.

```
$data List<class A> = Nil() | Cons(A a, List<A> b) ...
```

But then `Nil` and `Nil__` would be classes in the translation and `Nil()` would be the handle class (nullary) ctor which according to the default skeletons would apply operator `new` when applied to yield a new body object. As body objects according to the provided runtime library require at a minimum space for a virtual function table pointer and reference counter, the net effect would be that applications of `Nil<A>()` for any type `A` would result not in some null object corresponding to the usual notion of "nil" but a real space consuming body object.

The `Cons` sum component for the `List` example has its own body-user-code in which two "setters" are defined and a custom print function. The setters provide update capabilities as by default the corresponding member variables in the body class `Cons__` are not `const` qualified. There is a way to specify `const` qualification, by specifying a 'const' accessor construct. If it was desired for example that `List<A>` should be immutable rather than mutable, the declaration could be as follows.

```
$data List<class A> = 0 | Cons(A a => const, List<A> b => const) ...
```

Note that interface immutability can also be achieved simply by not providing interface update functions; using the `const` accessor construct serves to emphasize both interface and implementation immutability. If in addition custom access was desired for `List<A>`, the declaration could be as follows.

```
$data List<class A> = 0 | Cons(A a => const (), List<A> b => const ()) {
  const A a() {return a_;} // user defined
  const List<A> b() {return b_;} // user defined
  ...
} : {
  ...
};
```

The accessor construct provides the means for specifying various member variable access mechanisms.

```
accessor
-> '=>' '(' ' ' )'           -- mutable user defined access function
-> '=>' 'const' '(' ' ' )'  -- immutable user defined access function
-> '=>' 'const'             -- immutable generated access function
->                          -- mutable generated access function (default)
```

The product-decl construct provides the means for specifying various ctor parameter passing mechanisms.

```
product-decl
-> 'const'? alpha-type-spec '&'? id
```

A product-decl translates directly to a ctor parameter declaration. For example a sum "Foo(const Bar& baz)" with product-decl "const Bar& baz" has a translated ctor that looks like "Foo(const Bar& baz) ...".

The product-decl construct does not apply to corresponding member variables. In particular there is no provision for specifying reference member variables either via the product-decl or accessor constructs. Alpha data types are in general instances of compound concrete data types – they contain other such types. Containment in this manner is an ownership rather than a referencing relation.

The sum grammar rule allows for optional specification of the 'explicit' keyword:

```
sum
-> 'explicit'? id products code?
-> '0'
```

The keyword has the same effect for generated itor handle class ctors that instantiate body objects as it has for generated base type ctors.

The root-code construct provides the means for extending (or defining) root classes. There is however a restriction as to what functions can be directly defined. The order in which translation classes are generated implies that there can be no direct references to itors from root classes, so functions are restricted accordingly. That is why List<A>::length() for example cannot be directly defined in the root class as part of the root-code construct (it is declared there but defined later).

The alpha-type-spec construct directly corresponds to the analogous C++ means for specifying references to named types that may have template arguments, noting that referenced types must be alpha types.

Alpha Unit Types

Alpha unit types are a special case of alpha data types, in that they have just one sum component, and are primarily devices of convenience because the name of the itor is the name of the type, eliminating the need to invent a new name for certain common cases. The grammar portion for alpha unit type declarations follows.

```
alpha-unit-type-decl
-> '$unit' 'explicit'? id alpha-type-head? products unit-code? ';'

unit-code
-> body-user-code? root-code?
```

The grammar rule allows for optional specification of the 'explicit' keyword, and has the same effect for generated handle class ctors that instantiate body objects as it has for generated base type ctors.

The beta.h Pair type provides a typical example.

```
$unit Pair<class A, class B>(A a, B b);
```

There is a special consideration regarding mutually recursive alpha typing contexts, further elaborated below.

Alpha Type Types

Alpha type types are more or less the app equivalent of typedefs. They have essentially the same renaming role as typedefs but renamed types are also alpha types. The grammar portion for alpha type type declarations follows.

```
alpha-type-type-decl
```

```
-> '$type' id alpha-type-head? '=' alpha-type-spec ';' 
```

An example without the alpha-type-head construct follows.

```
$type Atoms = List<Atom>;
```

This translates directly to a typedef.

```
typedef List<Atom> Atoms;
```

The alpha-type-head construct is provided even though the translation is at the present time not valid C++ (it is there "just in case").

```
$type Xs<class X> = List<X>;
```

The translation follows.

```
template <class X>
typedef List<X> Xs;
```

It is also possible to get an approximation of template typedefs using the `--skeletons=f` option when `f` contains something like the following.

```
$+ type_synonym
$temp_head$
struct $id$ : public $type_spec$ {
    $id$() : $type_spec$() {}
    $id$($type_spec$ arg) : $type_spec$(arg(), copyok) {}
};
$-
```

Given such a skeleton, the example above translates as follows.

```
template <class X>
struct Xs : public List<X> {
    Xs() : List<X>() {}
    Xs(List<X> arg) : List<X>(arg(), copyok) {}
};
```

Note that the `--skeletons=f` option is available via the `$option` syntactic construct, so it is possible to selectively generate appropriate translations through judicious use of the construct.

There is a little more to alpha type types than this apparent redundancy. In particular such types can be defined mutually recursive.

Alpha Forward Types

Alpha forward types provide for mutually recursive alpha types. The grammar portion for alpha forward type declarations follows.

```
alpha-forward-type-decl
-> '$data' id alpha-type-head? '$forward'? root-code? ';'
-> '$type' id alpha-type-head? '$forward'? ';' 
```

Consider again the declaration for `List<A>`.

```
$data List<class A> = 0 | Cons(A a, List<A> b) ...
```

The List alpha type is self recursive, but not mutually recursive. Self recursive types are about as natural as non-recursive types insofar as alpha typing goes (they require no special considerations beyond those already in place). However mutually recursive types are beasts of a different sort.

Suppose it is desired to represent the parse of an alpha-type-params syntactic construct as an abstract syntax tree (ast), and further that the ast structure is to be defined as an alpha type. Here's the grammar for the construct again.

```
alpha-type-params
-> '<' alpha-type-param/',' '>'

alpha-type-param
-> id alpha-type-params?
```

A first attempt might look something like this.

```
$type Alpha_Type_Params $forward;

$unit Alpha_Type_Param(Leaf id, Alpha_Type_Params atps);

$type Alpha_Type_Params = List<Alpha_Type_Param>;
```

This won't work because as it happens unit types cannot be declared within the scope of a mutually recursive alpha typing context – i.e. the contextual scope defined by an initial forward declaration and its corresponding resolving declaration. The reason for this is that the translation for unit types takes advantage of the reduced space of app generated names.

Instead of generating a root class, an itor handle class, and corresponding body class, app generates only a body class and itor handle class that also serves as the root class. But this means that there can be no self recursive references (i.e. from the body class to the root class), as such references assume predefinition of the root class. For the sake of apparent consistency, app precludes the declaration of unit types within the scope of mutually recursive typing contexts. (It is an anomaly that the same restriction does not apply to base types – but then base types are anomalous by nature).

The remedy for this situation is to use an alpha data type rather than a unit type, because the generation order of translation classes for data types does allow for such recursive declarations. This requires an artificial name for the singleton sum component. A rule-of-thumb is to derive it from the type name, using 'a_' or 'an_' as a name prefix.

```
$type Alpha_Type_Params $forward;

$data Alpha_Type_Param =
  an_Alpha_Type_Param(Leaf id, Alpha_Type_Params atps);

$type Alpha_Type_Params = List<Alpha_Type_Param>;
```

When an alpha data type or type type T1 is declared forward, and then another such type T2 is also declared forward but before the resolving declaration for T1, then T2 must be resolved prior to T1. In other words,

```
$data T1 $forward;
...
$type T2 $forward;
...
$type T2 = ...;
...
$data T1 = ...;
```

is okay, but

```
$data T1 $forward;
...
$type T2 $forward;
...
$data T1 = ...;
...
$type T2 = ...;
```

is not. The reason for this constraint is primarily aesthetic, and secondarily algorithmic.

App employs an algorithm for treating mutually recursive types that involves a "forward stack" that is pushed on forward declarations and popped on resolving declarations. It is best not to introduce any other (non-declaring) code that depends on alpha types still in the forward stack until after the stack empties as certain corresponding translation classes are not generated until after the stack becomes empty.

For mutually recursive alpha data types, the corresponding forward declaration allows a root-code construct that provides the means for extending (or defining) root classes. The same restrictions regarding function definitions described earlier for root classes applies here. The translation for forward data types includes root class generation for forward declarations but not for resolving declarations; app warns you if any resolving declaration has an associated root-code.

It should also be mentioned that there are no additional constraints or special considerations regarding parametric alpha types; they're first class with respect to recursion.

You may wish to examine the ast.h app source file as it contains the complete set of alpha type declarations for asts corresponding to the app input grammar.

Special Considerations

Int is a base type for "boxed" native ints provided in beta.h:

```
$base Int(int data) {
    bool print(ostream& os) {os << data(); return true;}
};
```

Notice the use of the constant '0' in the following expression.

```
Cons<Int>(0, 0)
```

The constant is being used in two ways here. First, for alpha base type Int app generates a corresponding handle class with a ctor argument of type int, and as the ctor is not explicit, 0 implicitly converts to Int(0). Second, for alpha type List<A> app generates a corresponding itor class Cons such that for any type A, the type of the second ctor argument is List<A>. For any alpha data type T, or base or unit type T without a nullary ctor, the default constructed object T() is defined to be the null object a.k.a. 0 (if T is a base or unit type that does have a nullary ctor, T() is not the null object). Hence the above expression example is equivalent to the following.

```
Cons<Int>(Int(0), List<Int>())
```

The default constructed object T() can be used instead of 0 in those cases when the use of 0 may either cause problems at compile time, or generate a possibly unexpected result at run time (provided that T() yields a null object). The latter case may arise for example when 0 is passed as an argument to a unary (single argument) ctor K, expecting 0 to denote a null argument of some alpha type A. But K(0) is equivalent to K(), making the

K object itself null. Instead use `K(A(0))`, or equivalently (if `A()` yields the null object), `K(A())`. The static `nil()` function can be used instead of the default ctor for handle classes with nullary ctors.

Generated handle classes for alpha types get a ctor with arguments of the following form (according to the default skeletons).

```
(Top__* handle, Copy docopy = ...)
```

The `docopy` argument is defaulted so the ctor can be unary and serve as an implicit conversion operator in certain cases, including those cases when `0` denotes the null object. This particular ctor form also has special uses relating to generated casting code, and to externally generated handles from e.g. yacc code.

The function `Top::zero()` provides another way to generate null alpha objects:

```
static Top__* Top::zero() {return 0;}
```

Thus the expression example above is also equivalent to the following.

```
Cons<Int>(Int(0), Top::zero())
```

Unary ctors can also cause potential problems given certain argument types. Consider again alpha type types. They are really just synonyms, so the following doesn't work (try it).

```
$type T1 $forward;
$type T2 = T1;
$type T1 = T2;
```

To break the circularity, one can introduce an intervening non-synonymous type, e.g.

```
$type T1 $forward;
$data T2 = U(T1 t1) | V(int x);
$type T1 = T2;
```

One might think then that `U(U(U(V(42))))` ought to be well-defined (it is) and produce something like `"U:{U:{U:{V:{42}}}"` as output. It doesn't – instead it produces `"U:{V:{42}}"`.

The problem involves overload resolution of unary ctors like `U`. The expression `U(V(42))`, while of type `T2` (or `T1`), is treated by the compiler as `"const Top&"` instead of `T2` (`T1`) when used as an argument for the `U` ctor. The `Top` copy ctor is favored for overload resolution, so `U(U...(U(V(42)))...)` winds up being equivalent to `U(V(42))`. (Or at least that's what the compiler used seemed to indicate). The following does produce the expected result.

```
T2 a = U(V(42));
T2 b = U(a);
T2 c = U(b);
cout << c << endl; // "U:{U:{U:{V:{42}}}"
```

The problem does not occur with non-unary ctors, because only unary ctors can serve as implicit conversion operators, and it does not occur when the type of a unary ctor argument is not identical to (or synonymous with) the associated class (accounting also for parameterization), because then copy construction cannot apply.

The problem can also be avoided by defining a constructor function as a static member of a problematic ctor, e.g.

```
$type T1 $forward;
```

```
$data T2 = U(T1 t1) {} {static T2 u(T1 t1) {return U(t1);}} | V(int x);
$type T1 = T2;
...
cout << U::u(U::u(U::u(V(42)))) << endl; // expected result
```

The same method may also be useful for avoiding some of the problems related to implicit conversion mentioned above.

To summarize, unary ctors can serve as implicit conversion operators, and copy ctors can serve as implicit (and possibly unexpected) identity operators, and so as usual one should be a little careful with them. It is possible to avoid identity operator problems by using static constructor functions for problematic ctors, and which may also be useful for some of the problems related to implicit conversion (the 'explicit' keyword may also be useful).

[CONTENTSNEXT](#)

Match Statements

Algebraic types are basically useless without some kind of corresponding pattern matching facility. The alternative would be explicit selection and projection functions that would in effect offer no more functionality for C++ than what is already provided by RTTI & dynamic casting. (This is not to say that app/applib is not useful without pattern matching, as app/applib support for the handle body idiom and reference counting can be considered useful.)

Match statements are the app way of doing pattern matching. When app encounters a match statement it is subjected to a translation process that produces corresponding C++ conditional statements. App just assumes that whenever such a statement is encountered, it appears in a context valid for C++ statements. If you specify such a statement in some other context, say in the middle of a declaration, app knows nothing of it and will happily generate conditional statements in the middle of the declaration anyway.

General Form of Match Statements

The app input grammar portion for match statements follows.

```

match-stmt
  -> '$match' subjects pattern-cases

subjects
  -> '(' balanced-items-list ')'

pattern-cases
  -> '{' pattern-case* '}'

pattern-case
  -> patterns guard? '=>' '{' balanced-item* '}'

patterns
  -> '(' pattern-list ')'

pattern-list
  -> pattern/','

pattern
  -> positional-pattern

positional-pattern
  -> inductive-pattern
  -> binding-pattern
  -> zero-pattern

inductive-pattern
  -> pattern-decl subpatterns

binding-pattern
  -> id

zero-pattern
  -> '0'

pattern-decl
  -> pattern-spec id?

pattern-spec
  -> id alpha-type-params?

```

```

subpatterns
  -> '(' subpattern-list? ')'

subpattern-list
  -> subpattern/','

subpattern
  -> positional-pattern
  -> nonpositional-pattern

nonpositional-pattern
  -> explicit-pattern
  -> implicit-pattern
  -> ellipsis-pattern

explicit-pattern
  -> id ':' positional-pattern

implicit-pattern
  -> id ':'

ellipsis-pattern
  -> '...'

guard
  -> '|' '(' balanced-item+ ')'

```

Here is a simple example taken from the beta.h library code.

```

template <class A>
int List<A>::length() const {
  int result = 0;
  for (List<A> xs = *this;;) {
    $match (xs) {
      (Cons<A>(_ , xs1)) => {
        result++;
        xs = xs1;
      }
      (0) => {break;}
    }
  }
  return result;
}

```

The corresponding "pretty" translation follows.

```

template <class A>
int List<A>::length() const {
  int result = 0;
  for (List<A> xs = *this;;) {

    //$$ begin $match
    { bool _alpha_fail = true;
      List<A> _alpha_norm_1 = xs;
      if (dynamic_cast<Cons__<A> *>(_alpha_norm_1())) {
        Cons<A> _alpha_cast_1(_alpha_norm_1());
        List<A> xs1 = _alpha_cast_1->b();
        _alpha_fail = false;
        //$$ begin user statement code

        result++;
        xs = xs1;

```

```

        //$$ end user statement code
    }
    if (_alpha_fail && _alpha_norm_1() == 0) {
        _alpha_fail = false;
        //$$ begin user statement code
        break;
        //$$ end user statement code
    }
    if (_alpha_fail) escape("match failed");
}
//$$ end $match

}
return result;
}

```

Subject Expressions

Each subject of a match statement is taken to be an arbitrary C++ expression. App generally normalizes subject expressions to identifier form – even if they are already in identifier form, by generating appropriate declarations as prolog code. One reason for doing this is that subjects generally appear more than once in the translation code, and so otherwise the same subject expression might be evaluated more than once (if not already in identifier form).

Another reason is for typing purposes. App attempts to infer subject types from pattern cases; it warns you if either no such type can be inferred or if an inferred type is alpha-type inconsistent, in which cases 'Top' becomes the type (which is technically correct but also a generalization). There are additional cases when 'Top' may (silently) become the type because there is no other choice – i.e. when '0' patterns occur in relative isolation, usually as a result of translating certain kinds of nested patterns ('0' patterns match when the associated subject value is null).

The type inferencing algorithm app uses is pretty simple. For the example above the pattern case:

```
(Cons<A>(_, xs1)) => {...}
```

directly implies a subject type `List<A>`. The general idea is that the sum components of an algebraic type directly imply the type. As the patterns of a given pattern case directly refer to sum components (e.g. `Cons<A>`) it is a simple matter to determine the corresponding type (e.g. `List<A>`). Type parameters appearing in a pattern are taken as instances of corresponding declaration parameters. For example a pattern case like:

```
(Cons<int>(_, xs1)) => {...}
```

implies type `List<int>` because "int" is taken as an instance of "A" in the declaration:

```
$data List<class A> = 0 | Cons(A a, List<A> b) {...};
```

It may be the case that an inferred type is inconsistent with contextual type. For example a subject expression of contextual type `int` but with pattern cases implying type `List<int>` results in an inconsistent type that app cannot detect (it knows nothing of context), and in such cases app lets the compiler complain about it (having no other choice).

Normalizing subject expressions may result in some code redundancy (which however an optimizing compiler ought to be able to handle), but it does allow pattern matching to be first class with respect to static typing.

Pattern Matching Translations

Match statement pattern cases are translated in appearance order. Match statements translate to an ordered sequence of conditional 'if' statements corresponding to the pattern cases and wrapped as a single compound statement. If it is possible for a match statement to fail, epilog code is generated in the form of a final 'if' statement that checks for failure and which calls the alpha core function 'escape' in that case (see also `alpha.h`).

A patterns construct is an ordered sequence of pattern constructs corresponding to an ordered sequence of subjects. Such patterns are also called top-level patterns, particularly when it is useful to distinguish between them and nested i.e. subpattern constructs.

There are three general classes of patterns, inductive, binding, and zero. Inductive patterns refer to an alpha base, unit, or data type (via the id part of a pattern-spec construct), and may have subpatterns that inductively refer to such types (except for those involving base type references). Inductive patterns match corresponding subjects if the subjects are compatibly constructed and the subpatterns match. Binding patterns always match corresponding subjects, and zero patterns match null valued subjects.

Patterns can further be classified as positional or nonpositional. Positional patterns correspond with subjects according to sequence order. Nonpositional patterns are defined in terms of positional patterns, and are further discussed below; patterns are understood to be positional unless otherwise indicated. Top-level patterns can only be positional.

The general translation scheme for inductive patterns involves the generation of selection code as conjunctions of `dynamic_cast` expressions over polymorphic body class objects via operator `()` applied to subjects, casting code for body to handle class hierarchy mapping and cross/downcasting, and projection code to access member variables via operator `->`. Zero patterns involve generated null-testing selection code, and binding patterns involve generated casting and projection code.

Inductive Patterns

Patterns containing subpatterns are inductive patterns, and are inductively translated by synthesizing new subjects with values associated with non-binding subpatterns, taking the subpatterns as new (top-level) patterns, the whole being considered as an implicit nested match statement. When all such nesting is accounted for, and if there is a guard portion, taken as an arbitrary C++ expression, the guard becomes the conditional part of a final innermost 'if' statement. The balanced-item* construct associated with the pattern-case construct, taken as an arbitrary C++ statement or statement sequence, is then emitted as the then-part of the final 'if' statement as statement code. If there is no guard, the statement code is emitted directly.

The inductive-pattern construct has the same basic syntax used for applying a constructor. The syntax `"Cons<int>"` for example is used both for constructing an object of (alpha) type `List<int>`, as in the application `"Cons<int>(x, y)"`, and for specifying a pattern intended to match such objects as subjects, as in the pattern `"Cons<int>(x, y)"`. The above application results in an anonymous object, but in declarative contexts e.g. `"Cons<int> xs(x, y)"` the object is named (`xs`). The id part of the pattern-decl construct provides for analogous naming – it stands for the value of the corresponding subject.

The id part of a pattern-spec construct presumably mentions either a base type, unit type, or itor sum component of a data type; app complains otherwise. In particular app complains if a type type is mentioned. While it would be technically feasible for app to transitively dereference type types, thereby allowing type types to be so mentioned, app refuses to do so on the grounds that type types used in that manner could too easily undermine the underlying intent. Consider for example the following.

```

$type Point = Pair<int, int>; // a point on a plane defined by coordinates
$type Edge = Pair<int, int>; // an edge of a graph defined by vertices
...
$match (foo) {
  (Point(_, _)) => {...}
  (Edge(_, _)) => {...}
}

```

If alpha type types could be so mentioned, subject `foo` would be (alpha) type consistent, yet nevertheless there is clearly something wrong with the intent here. Alpha type types are renaming devices, primarily of convenience. If it is truly necessary to refer to the same underlying thing, as patterns, and in different ways, use unit or data types instead.

The utility of alpha base types for pattern matching purposes is limited. Any and all subpatterns of pattern constructs in which they are mentioned as a pattern-spec id must be binding patterns; `app` complains otherwise. This is because elements of a base-products construct can involve arbitrary C++ types and `app` has no idea how to match subjects of such types as anything other than binding patterns.

Base types introduced in order to make arbitrary C++ types appear as alpha types (via the "`$user {} $user{}`") construct) should probably not be referenced as a pattern-spec id at all, due to the various assumptions that generated selection, casting, and projection code requires. In short, only binding patterns should be used for matching such types. Examples:

```

// this is okay:
$base string;
string s = "okay";
$match(Cons<string>(s, 0)) {
  (Cons<string>(t, _)) => {cout << t << endl;}
  (_) => {}
}

// this is not - app accepts it but generates uncompileable code:
$base float(float x) $user {} $user{};
float y = -42;
$match (y) {
  (float(z)) => {cout << z << endl;}
}

```

Binding Patterns

Binding patterns correspond to generated variables with the same id name, and when appearing as subpatterns, are bound to corresponding subject components in the translated code by means of generated casting and projection code that appears immediately after the generated selection code in the constraining conditional expression. Top-level binding patterns are bound directly.

Binding patterns with ids spelled `'_'` are taken as wildcard patterns – they match anything and in themselves result in no generated code.

Binding patterns must be unique for each pattern case (excepting wildcards); `app` complains otherwise. In a pattern like `"Foo(x, x)"` for example the binding pattern `"x"` is not unique. Supposing that binding patterns need not be unique, the implied effect of nonunique binding patterns would be application of operator `==`. One way to get that effect with unique patterns is to use a guard involving operator `==` as in `"(Foo(x1, x2)) | (x1 == x2)"`.

Zero Patterns

Zero patterns are intended for use with corresponding subjects that may be nominally null and alpha typed such that a '0' sum component appears in the associated alpha data type declaration; app warns you otherwise. The bool functions `Top::null()` and `Top::not_null()`, and `Top__ * Top::operator ()` may be utilized to test for null cases in general.

Parametric Pattern Types

Generated projection code involves the substitution of type parameters (alpha-type-param) in pattern-spec constructs for corresponding declaration parameters (alpha-type-head-param construct). Given for example:

```
$data List<class A> = ...
```

and:

```
$match (ys) {
  (Cons<int>(x, xs1)) => {...}
}
```

the generated code involves the substitution of `int` for `A`.

Parametric alpha base types cannot participate in such substitutions, as app doesn't know what to do about it, and complains if given a match statement that suggests otherwise. Inductive patterns therefore cannot involve parametric base types; you can always use accessor functions instead.

Nonpositional Patterns

A pattern of the form "`x:y`" is a nonpositional (explicit) pattern, where `x` is an alpha unit or data type product-decl id and `y` is a positional pattern that may contain nonpositional subpatterns. The id "`x`" must be associated with the alpha unit or data type to which the inductive pattern that contains the given subpatterns refers. Only subpatterns can be nonpositional.

A nonpositional (implicit) pattern of the form "`x:`" is taken as shorthand for the nonpositional explicit pattern "`x:x`". A nonpositional (ellipsis) pattern of the form "`...`" is taken as "any other pattern not explicitly mentioned is '`_`' i.e. a wildcard".

If any subpattern appearing in a subpattern-list is nonpositional, all such subpatterns must be nonpositional. An ellipsis pattern should appear last in the list if the number of subpatterns is less than the number implied by the associated product-decl; app warns you otherwise.

App internally rewrites nonpositional patterns as equivalent positional patterns. Here's an example.

```
$data A = B(int c, Atom d, A e) | F();
...
A a = B(0, "ok", B(0, "ok", F()));
...
$match (a) {
  (B(e:B(d:g, c:, ...), ...)) => {...c...g...} // internally rewritten as:
  (B(_, _, B(c, g, _))) => {...c...g...}
  ...
}
```

Runtime Type Selection

By default app generates runtime type selection (stor) code that involves `dynamic_cast`. Considering again the `List<A>::length()` function illustrated above, the match statement:

```
$match (xs) {
  (Cons<A>(_, xs1)) => {
    ...
  }
  (0) => {break;}
}
```

has generated code that involves `dynamic_cast`:

```
//$ begin $match
{ bool _alpha_fail = true;
  List<A> _alpha_norm_1 = xs;
  if (dynamic_cast<Cons__<A> *>(_alpha_norm_1())) {
    ...
  }
  if (_alpha_fail) escape("match failed");
}
//$ end $match
```

This behavior can be changed either globally, via a command line option, or locally on a match statement basis, via the `alpha-nested-item` option construct. The option in either case is `'--stor_method=m'` (or `'-sm=m'`) where, if `m=d` the generated code involves `dynamic_cast` as before, and if `m=r` the code involves an internal alpha core function `"raw_type_is"` that essentially just does string comparisons between "raw" types – i.e. handle class names without parametric elaboration (see also `alpha.h`). When used locally the option construct applies to subsequent match statements as textually encountered by app, the option being effective until the next such construct is encountered; e.g.

```
$option "-sm=r"
$match (...) { // involves raw_type_is
  (...) => {
    ...
    $option "-sm=d"
    $match (...) {...} // involves dynamic_cast
    ...
  }
}
$option "-sm=r"
$match (...) {...} // involves raw_type_is
```

For the `List<A>::length()` example the generated code with `'-sm=r'` looks like:

```
//$ begin $match
{ bool _alpha_fail = true;
  List<A> _alpha_norm_1 = xs;
  if (raw_type_is("Cons",_alpha_norm_1())) {
    ...
  }
  if (_alpha_fail) escape("match failed");
}
//$ end $match
```

This code is generally faster than `dynamic_cast` code. Use of the `raw_type_is` function requires that associated app generated body classes define function `"raw_type"` that overrides virtual `"Top__::raw_type"` (`alpha.h`). These functions are automatically generated by app for unit and data types but not base types, and

all just return a related string, e.g. "Cons" (pattern matching involving base types always involves generated `dynamic_cast` code, a non-issue when base type accessor methods are used instead of pattern matching).

This works in general because pattern matching is statically typed; for any given match statement involving a subject of alpha type T, all associated patterns corresponding to the same sum component of T have the same "raw" form, provided that T is consistent (app and the C++ compiler ensure this). In the example above only `Cons<A>` patterns can validly appear in the match statement as inductive patterns for any given A, so `raw_type_is("Cons",...)` suffices, and that is the case in general.

The use of the `raw_type` and `raw_type_is` functions is safe to the extent that static typing in C++ is safe. It is still possible to externally construct bad alpha typed instances e.g. via bad yacc code, but then there is a larger problem (in which cases option `'-sm=d'` might be useful).

The functions are not used in generated code by default; app generated applib code (e.g. `_beta.h`) as provided involves `dynamic_cast`; you will need to re-app the relevant files (e.g. `beta.h`) with option `'sm=r'` if you want to use the applib code together with that option and option `'-ori'` (i.e. if you want to include the provided app generated applib headers with option `'sm=r'` applied).

Case Coverage

App employs a simple analysis algorithm to determine if all relevant sum components are mentioned in the pattern cases of a given match statement, and warns you if something is missing, unless the last pattern case patterns are all binding patterns (including wildcards `'_'`), as such pattern cases in effect constitute default cases.

Despite the simplicity of the algorithm, it is an important part of app, as it directly addresses one of the primary objections against the casing style itself – how do you deal with changes to the underlying case structure in a way that ensures the programmer is not left completely in the dark? The traditional casing style tends to induce severe maintenance related problems, especially for projects involving multiple programmers. Add a new case here, delete one there, and the next thing you know you have a lot of code not accounting for it, and no way to deal with it except by grepping and/or testing.

Cases in the present system correspond to sum components. If you delete or change the (itor) name of a sum component, app and/or the compiler will detect potential problems thereby introduced – case symbols are static entities that both app and the compiler track. If you add a new sum component, and fail to account for it in some match statement (without a default case that is), app lets you know about it.

In a sense default cases correspond to default virtual functions of a base class corresponding to the root class of some alpha data type, and non-default cases correspond to overrides of virtual functions defined in derived classes corresponding to the associated injectors. Traditional OO wisdom has it that defaulted virtual functions can be dangerous, as in effect they provide (possibly unexpected) behavior for cases not explicitly accounted for. The same wisdom applies here. If you have default cases, app has no idea whether that is what you intended or not; default cases effectively short-circuit the case coverage analysis algorithm.

Default cases however tend to be convenient, and also allow for more efficient generated code because without them app generates match statement epilog code that checks for match failure (omitted if a default case appears). To allow for the convenience and implied efficiency of default cases, while at the same time still providing for case coverage analysis, app lets you specify "safety cases". A safety case is any pattern case that follows another pattern case in which all patterns are binding patterns, i.e. any pattern case that follows a default case.

In effect safety cases prevent default cases from short-circuiting the analysis algorithm. Pattern cases

following a default case are treated as irrelevant at run time and are excluded from the translation. App warns you if a safety case has associated guard or statement code. Here's an example.

```
$data A = B(int c, Atom d) | E(int f) | G(Atom h, int i);  
...  
A a = B(0, "ok");  
...  
$match (a) {  
  (B(x, y)) => {...}  
  (_) => {} // default case  
  (E(...)) => {} // safety case  
  (G(...)) => {} // safety case  
}
```

[CONTENTSNEXT](#)

The Runtime Library

The runtime library `applib` provides several components for app generated programs, an 'alpha' core component, an alpha type utility component 'beta', and a few additional utility components.

Most library component symbols are namespace qualified according to the `#define` symbol `APP_LIB_NS`, which by default is defined as 'applib', and which also has by default the synonym 'alpha' – see also the `applib_config.h` file. All alpha types, and overloaded operators parametrically depending on such types, must be defined in the `applib` namespace. Additionally this namespace must be introduced prior to using any such types from the context of app generated match statement code, by either a using directive or appropriate using declarations, as app itself knows nothing of namespaces.

Alpha Component

`Applib` provides a core component called `alpha`, the functionality and interface of which is required by app generated programs. It defines the two alpha base classes `Top` & `Top__`, and a few auxiliary types and functions.

In addition to `Top` & `Top__`, the core component provides the type `Copy`, a replacement for type `bool` used for reference counting control purposes, and a customizable class `Alpha_Control` that serves primarily as a wrapper class for output and error handling. Alpha types support stream output via the `<<` operator; the `Alpha_Control` class can be used to customize its behavior.

A number of functions are forwarding interfaces for members of `Alpha_Control`; error handling: `escape`, `escape_output`; output formatting: `form_type_name`, `form_using_zero`, `form_using_rtti`, `form_class_part`, `form_param_part`, `form_under_part`, `form_final_part`, `show_begin`, `show_delim`, `show_end`, `show`, `showln`; utilities: `write_string`. There are also a couple of string related utility functions: `dup_string`, `new_string`.

It should be noted that this core component is provided as a kind of default implementation rather than as an absolute requirement. In other words you may define your own core if so desired, and in fact you are not required to use the provided runtime library at all, though if the provided core is not used, you must define a core with similar functionality and consistent interface, to the extent required to satisfy the relevant assumptions of app generated code. See the `alpha.h` and `alpha.cpp` library files for additional information.

Beta Component

`Applib` provides a component called `beta` that defines a number of simple and common alpha types as a standard library of sorts, and some additional related functions and classes. All of the alpha types therein have all comparison operations defined for them, and are (or should be) directly usable as STL contained types.

The alpha type declarations include the following.

- certain native types: `bool`, `char`, `short`, `unsigned`, `int`, `long`, `float`, `double`
- certain boxed native types: `Int`, `Long`, `Float`, `Double`
- simple strings: `Atom`
- parametric mutable tuples: `Pair`, `Triple`
- parametric mutable singly linked lists: `List`
- parametric singleton sets: `Single`

The native types are base types declared via the (implied) `"$user {} $user {}"` construct and have no corresponding generated classes. The boxed versions are base types that do have generated `handle/body`

classes (body object == "box"). In most cases the unboxed native versions can be used rather than the boxed versions. The boxed versions may be useful in circumstances when alpha typed objects must be uniformly represented (with the usual handle/body classes).

There is a fairly rich set of list related utility functions provided, most of which were explicitly designed to be consistent with corresponding functions of the [Pizza](#) language.

The essential List access functions are head() and tail(). List objects are constructed from Cons injectors and also null objects; the Cons iterator is of form Cons(A a, List<A> b), where a is the head and b is the tail. A non-null list is a sequence of Cons objects such that the last Cons object has a null tail. The provided List<A> member functions include the following.

```
A head() const; // first Cons product of this list
A head(A new_head); // setter
List<A> tail() const; // second Cons product of this list
List<A> tail(List<A> new_tail); // setter
int length() const; // number of elements of this list
List<A> take(int n) const; // first n elements of this list
List<A> drop(int n) const; // all but the first n elements of this list
A at(int i, bool escapes = true) const; // i'th element of this list, i>=0
A last() const; // last element of this list
List<A> reverse() const; // elements of this list in reverse order
List<A> append(List<A> ys) const; // this list with ys appended
List<A> extend(List<A> ys); // this list with ys appended destructively
List<A> filter(bool f(A)) const; // elements x of this list s.t. f(x) is true
int index(A y) const; // least i s.t. (*this)[i] == y, -1 if no such i
List<A> diff(List<A> ys) const; // this list without the elements of ys
List<A> diff(A y) const; // this list without the element y
List< Pair<A, A> > zip(List<A> ys) const; // this list and ys pairwise
List<A> sort_insert(A e) const; // this list with e inserted ordered by <
bool contains(A y) const; // index(y) >= 0
List<A> concat(A y) const; // append(Cons<A>(y, 0))
List<A> extend(A y); // extend(Cons<A>(y, 0))
A operator [] (int i) const {return at(i, false);}
List<A> init() const {return take(length() - 1);}
List<A> concat(List<A> ys) const {return append(ys);}
Pair< List<A>, List<A> > split(int n) const {
    return Pair< List<A>, List<A> >(take(n), drop(n));
}
```

There are also a number of list related template functions.

```
template <class A> // the concatenation of all elements of xss
List<A> concatenate(List< List<A> > xss)

template <class A, class B> // apply f to each element of xs
void forall(B f(A), List<A> xs)

template <class A, class B> // list of f applied to each element of xs
List<B> map(B f(A), List<A> xs)

template <class A, class B> // catenation of f applied to each element of xs
List<B> bind(List<B> f(A), List<A> xs)

template <class A, class B> // f applied to each element of xs from left
B foldl(B f(B, A), B z, List<A> xs)

template <class A, class B> // f applied to each element of xs from right
B foldr(B f(A, B), B z, List<A> xs)
```

```
template <class A> // the list of n x's
List<A> repeat(A x, int n)
```

The Single alpha type is a convenient way for example to represent in asts some kinds of optional syntactic constructs.

```
$data Single<class A> = 0 | Single_Element(A a) : {
  bool empty() const {return null();} // true if zero constructed
  bool not_empty() const {return not_null();} // false if zero constructed
  A element() const; // a if Single_Element(A a) constructed
};
```

An ordinary C++ template class Associative_List is included that provides simple linear multimap-like services.

```
template <class A, class B>
class Associative_List {
public:
  typedef List< Pair<A, B> > State;
  Associative_List() {}
  Associative_List(State list) : list(list) {}
  virtual ~Associative_List() {}
  virtual State state() const {return list;}
  virtual State state(State new_list) {return list = new_list;}
  virtual void init() {list = 0;}
  virtual B get(A x) const; // the first value with key == x
  virtual List<B> getall(A x) const; // the list of all such values
  virtual List<B> getall(A x, List< Pair<A, B> > z) const; // prepended to z
  virtual bool has(A x) const; // true if there is at least one key x
  virtual void addnew(Pair<A, B> p) {init(); add(p);}
  virtual void addnew(A x, B y) {addnew(Pair<A, B>(x, y));}
  virtual void add(Pair<A, B> p) {list = Cons< Pair<A, B> >(p, list);}
  virtual void add(A x, B y) {add(Pair<A, B>(x, y));}
protected:
  State list;
};
```

The class also appears as an alpha type:

```
$base Associative_List<class A, class B>;
```

Another ordinary C++ template class Extending_List is included and provides for efficient repeated destructive appending list operations, with additional related functions that yield stack/queue/deque capabilities.

```
template <class A>
class Extending_List {
public:
  Extending_List() : triple(State(0, 0, 0)) {}
  Extending_List(List<A> xs) : triple(State(0, 0, 0)) {extend(xs);}
  Extending_List(A x) : triple(State(0, 0, 0)) {extend(x);}
  virtual ~Extending_List() {}
  virtual void extend(List<A> xs); // add last elements, subsequently mutated
  virtual void extend(A x) {extend(Cons<A>(x, 0));} // add last element
  virtual void prepend(A x); // add first element
  virtual void retract(); // remove last element
  virtual void consume(); // remove first element
  virtual A last(); // last element
  virtual A first(); // first element
  virtual length() {return triple.third();} // list length
  virtual List<A> list() const {return triple.first();} // the list
```

```
protected:
    typedef Triple<List<A>, List<A>, int> State;
    State triple;
};
```

The class also appears as an alpha type ala `Associative_List`.

Additional Components

AppLib also provides miscellaneous additional components, consisting of a small number of utilities and the like that may be found useful. The library can be configured in various ways, excluding certain parts, so as to make various lean versions if so desired.

To be useful as a normal null-terminated string, taking `str()` of an `ostrstream` object with a dynamic buffer first requires 'ends' then unfreezing the internal buffer frozen by `str()` in order for `~ostrstream` to release it. The provided `Str` class automates this behavior.

```
class Str : public ostrstream {
public:
    Str(bool unfreezes = true) : unfreeze(unfreezes) {}
    virtual ~Str() {}
    virtual const char* string(bool end = true) {
        if (end) *this << ends;
        const char* s = str();
        if (unfreeze) rdbuf()->freeze(0);
        return s;
    }
protected:
    bool unfreeze;
};
```

The `Str` class, in combination with alpha type `Atom`, serves very well for many purposes that might otherwise be done using the standard C++ string class. A typical situation might involve an auto `Str` object that is built up by `<<` operations and then finally used to define an `Atom` via `Str::string()`. The app source itself makes extensive use of the combination.

Note that the default behavior for both `Str` and `Atom` is to use independent dynamic buffers – this is safe but less efficient than letting `Atom` reuse the `Str` buffer via ownership transfer, possible by constructing `Str` with a false 'unfreezes' argument, and `Atom` with a false 'clone' argument. Such sharing of course is best done only when it is clear that it is indeed safe to do so. A good practice is to construct both in close textual proximity, e.g.

```
Str s(false); // let Atom own the buffer
do_something_with(s);
Atom a = Atom(s.string(), false); // last use of s, a now owns the buffer
```

A class `Pretty_Printer` provides basic pretty printing capabilities for alpha types. The class customizes the `alpha.h` core class `Alpha_Control`. Note that it does not customize `Alpha_Control` by default. See the `pretty.h` file.

A class `Flow` provides simple debugging and tracing capabilities using a context stack pushed by the ctor and popped by the dtor, and which is customizable. See the `flow.h` and `flow.cpp` files.

Files `shared.h` & `_shared.h` contain classes that provide general non-intrusive reference counting capabilities. The `shared.h` file is an app source; the `_shared.h` file is the app generated version. Usage is akin to `std::auto_ptr` except that there is no ownership transfer mechanism – reference counted pointer ownership is a

shared rather than unique relation.

For further details see the sources in the applib directory.

Symbol List

This is a summary list of applib symbols directly exposed via #includes. Unless otherwise indicated, all are in the applib namespace.

Symbols in alpha.h excluding overloaded operators.

- Copy
- nocopy
- copyok
- Top__
- Top
- Alpha_Control
- escape
- escape_output
- form_type_name
- form_using_zero
- form_using_rtti
- form_class_part
- form_param_part
- form_under_part
- form_final_part
- show_begin
- show_delim
- show_end
- show
- showln
- write_string
- dup_string
- new_string
- raw_type_is

Symbols in beta.h & _beta.h excluding overloaded operators & body classes e.g. Int__ etc. (Top & Top__ appear in alpha.h, not _beta.h; Top also appears in beta.h as an alpha type declaration; the bool, char etc. symbols are alpha type declarations).

- bool
- char
- short
- unsigned
- int
- long
- float
- double
- Top
- Int
- Long
- Float

- Double
- Atom
- Pair
- Triple
- List
- Cons
- concatenate
- forall
- map
- bind
- foldl
- foldr
- repeat
- Single
- Single_Element
- Associative_List
- Extending_List

Symbols in flow.h, pretty.h, shared.h, str.h, yystype.h. (YY is not in the applib namespace).

- Flow
- Pretty_Printer
- Basic_Shared_Pointer
- Shared_Pointer
- Str
- YY

All visible #define symbols are prefixed 'APP_', except 'alpha' and 'YYSTYPE'.

[CONTENTSNEXT](#)

Hacker's Guide to the Source

Don't Panic!

App is written in more or less generic but relatively modern C++ (and itself of course). In particular it utilizes namespaces, RTTI, (simple) templates, and some STL containers (but not exceptions). It should be easy enough to port to just about any platform that supports a relatively modern C++ compiler.

The app code now runs under both MS VC++ 5.0 and g++ 2.91.66. I have isolated most (if not all) known platform specific constructs by using `#ifdef WIN32` and `#ifdef GCC` constructs, and by using files `app_platform.h` and `applib_platform.h` in the `app` & `applib` directories that localize platform specific includes etc. Note that `GCC` is presently defined if `WIN32` is not (see `app_platform.h`). There are also a number of `#ifdef GCC_BUG` constructs that hopefully will go away once g++ iostream support sufficiently improves (`GCC_BUG` is defined if `GCC` is).

The free (beta) cygnus cygwin software works pretty well for providing certain Unix-like utilities for Windows; see <http://sourceware.cygnum.com>.

Physical Structure

There are a number of directories under the top-level `alpha` directory that serve to isolate the major software components. The `app` directory contains the source for most of app itself; the directories `app_parser` and `app_scanner` contain the source for the parser and scanner components, organized as static libraries linked with the main code in the `app` directory.

There are a couple of subdirectories under `app` that I use for testing purposes. The `test` subdirectory is for sanity checking stuff. The `prev` subdirectory is used for fixpoint testing; typing 'make fix' will check that app is still able to regenerate itself. Changing the code enough for fixpoint testing suggests that you best have a stable previous version stashed away somewhere first. See the makefile for details.

The `applib` directory contains the runtime library code. Note that app depends on the library (for self-defining reasons). The `skel` directory contains the code for a small standalone program ("skel") used to generate a compilable header file (`skeletons.h`) containing class translation skeletons from the associated input file `skeletons.txt`. The `docins` directory also contains a small standalone program ("docins") for generating `.html` and `.txt` files in the `doc` directory from a set of specially marked-up `.txt` files.

I use flex and bison for the app scanner and parser components. You will need to use something similar if you change either. All major components live in their own files including most classes. Most symbols live in a namespace, of which there are several.

Logical Structure

App defines a set of framework classes that coordinates post-initialization activity by means of singleton objects established at initialization; see also the `framework.*` and `driver.*` files. The framework establishes four major code categories: scanner, screener, parser, and interpreter.

One of the underlying code philosophies is that it should be easy to use standard C scanners and parsers (e.g. flex & bison) even though the asts the parser generates are instances of alpha types. See the `alpha.h` and `c_*.c` files for details about interfacing reference counted ast objects with standard C code (it's a bit tricky but not too bad).

The general post-initialization control flow can be summarized as follows.

```
(parser calls screener calls scanner for each token)
every time the parser recognizes an alpha construct it calls interpret
interpret dispatches Alpha_Type_Decl or Alpha_Nested operator ()..
Alpha_Type_Decl operator () handles alpha type declarations
Alpha_Nested operator () dispatches Match or Option operator ()..
Match operator () handles match statements
Option operator () handles embedded $option constructs
```

Customizing

App provides a factory class 'Make' that can make it easy to customize the code without having to modify in many cases or recompile in some cases the original code. Most internal classes (except those involving self-defining alpha types) have associated construction functions as virtual members of the Make class, and which app calls upon to create related objects.

To customize such an internal class, specialize by derivation both the internal class to be customized, and the Make class by overriding the relevant virtual construction function so as to create an object of the specialized internal class; then arrange to define the singleton Make* factory::make object as an instance of the specialized Make class prior to app initialization – i.e. prior to main calling Driver::app (see also main.cpp).

It should only be necessary to modify the existing main.cpp file to establish the new factory class, then recompile the app code and relink with the custom code which presumably lives in its own directory, perhaps as a static library. Once the app code has been recompiled with the new main.cpp, it should only be necessary to relink it thereafter, provided that changes to the existing app code files are confined to main.cpp.

The scanner and parser components involve generated C code, so to customize that code it is in general necessary to build new static libraries for them, in addition to specializing the Make class, and possibly the framework classes A_Scanner and A_Parser, if needed to handle the new scanner or parser; the existing framework specializing classes Scanner and Parser may also be specialized. Some of the code in the app directory includes headers from the app_parser and app_scanner static library directories, so place the new versions in the same directories after having saved away the originals.

Special Considerations

One thing you really have to watch out for is changing the code in such a way that app is no longer able to regenerate itself. It is not so bad if you can use a previous stable version to patch things up but that is not always the case (of course at least theoretically you can always completely undo whatever changes fowled things up in the first place, but I'm talking about dealing with the changes rather than undoing them).

If the nature of the changes is such that you are tempted to back-propagate the changes into a stable previous version, you almost certainly shouldn't. One situation where this might crop up is when you change a skeleton or the core alpha.* library code in such a way that invalidates a skeleton. Rather than going back to a previous version and trying to patch up its skeleton or alpha.* files (which might entail going to a previous-previous version... etc. etc.), try the `---skeletons=f` option. It lets you dynamically load skeletons in a way that overrides the default skeletons as defined by the compiled skeletons.h file.

In any event, and as a general rule, it is best to introduce those kinds of changes small portions at a time if at all possible.

Projects

App is not done. If you have an idea & want to do something about it, please do. Also, please let me know; I am willing to consider folding such code into my code base if you like. You might also want to check the app home page before starting a new project, as I intend to maintain an up-to-date status report there regarding such matters. Here then are some project ideas.

Implement an efficient allocator – see the alpha.* files.

Provide object input not just output – see the beta.h file. Input should be consistent with output.

Turn the beta.h List<A> type into an STL compatible container class. Such a class could also use as a basis the current Extending_List<A> class in beta.h as it already provides much of the required functionality.

Handle generic objects, in a manner akin to how void* pointers can be used to reduce extraneous template expansions. I'm not convinced though that alpha types are all that problematic in this way – who really cares if there are 23 versions of List<A>::length() anyway? (In my Pascal days I once thought that's the way you had to do it). But supposing it's worth it, one way to do generics would be to introduce an intermediary class T_ between the handle/body classes T and T__, as illustrated below (a \$generic app keyword also seems needed).

```
class GPair__ : public Top__ {
public:
    GPair__(Top a, Top b) : a_(a), b_(b) {}
protected:
    Top a_;
    Top b_;
};

template <class A, class B>
class GPair_ : public GPair__ {
public:
    GPair_(A a, B b) : GPair__(static_cast<Top>(a), static_cast<Top>(b)) {}
    A a() {return A(a_());}
    B b() {return B(b_());}
};

template <class A, class B>
class GPair : public Top {
public:
    GPair() : Top() {}
    GPair(A a, B b) : Top(new GPair_<A, B>(a, b), nocopy) {}
    GPair_<A, B> *operator->() const {
        return static_cast<GPair_<A, B> *>(handle);
    }
    GPair_<A, B> &operator*() const {
        return static_cast<GPair_<A, B> &>(*handle);
    }
    static GPair<A, B> nil() {return GPair<A, B>(0, nocopy);}
};
```

The App Input Grammar

This is the grammar specification for app inputs.

Notation.

<code>x?</code>	zero or one <code>x</code>
<code>x*</code>	zero or more <code>x</code>
<code>x+</code>	one or more <code>x</code>
<code>x/y</code>	one or more <code>x</code> 's separated by <code>y</code>

Rules.

```

goal
  -> item*

item
  -> alpha-item
  -> any C++ token

alpha-item
  -> alpha-type-decl
  -> alpha-nested-item

alpha-type-decl
  -> alpha-forward-type-decl
  -> alpha-base-type-decl
  -> alpha-unit-type-decl
  -> alpha-data-type-decl
  -> alpha-type-type-decl

alpha-forward-type-decl
  -> '$data' id alpha-type-head? '$forward'? root-code? ';'
  -> '$type' id alpha-type-head? '$forward'? ';'

alpha-base-type-decl
  -> '$base' 'explicit'? id alpha-type-head? base-products code? ';'
  -> '$base' id alpha-type-head? ';'

base-products
  -> '(' balanced-items-list? ')

alpha-unit-type-decl
  -> '$unit' 'explicit'? id alpha-type-head? products unit-code? ';'

alpha-data-type-decl
  -> '$data' id alpha-type-head? '=' sums root-code? ';'

sums
  -> sum/'|'

sum
  -> 'explicit'? id products code?
  -> '0'

alpha-type-type-decl
  -> '$type' id alpha-type-head? '=' alpha-type-spec ';'

products
  -> '(' product-list? ')

```

```
product-list
  -> product/','

product
  -> product-decl accessor

product-decl
  -> 'const'? alpha-type-spec '&'? id

accessor
  -> '=>' '(' ' ' ')'
  -> '=>' 'const' '(' ' ' ')'
  -> '=>' 'const'
  ->

user
  -> '{' balanced-item* '}'

user-code
  -> '$user'? user

body-user-code
  -> user-code

handle-user-code
  -> user-code

code
  -> body-user-code handle-user-code?

root-code
  -> ':' handle-user-code

unit-code
  -> body-user-code? root-code?

alpha-type-head
  -> '<' alpha-type-head-param/',' '>'

alpha-type-head-param
  -> 'class' id

alpha-type-spec
  -> id alpha-type-params?

alpha-type-params
  -> '<' alpha-type-param/',' '>'

alpha-type-param
  -> id alpha-type-params?

alpha-nested-item
  -> option
  -> match-stmt

option
  -> '$option' string-literal

match-stmt
  -> '$match' subjects pattern-cases

subjects
  -> '(' balanced-items-list ')'
```

```
pattern-cases
  -> '{' pattern-case* '}'

pattern-case
  -> patterns guard? '=>' '{' balanced-item* '}'

patterns
  -> '(' pattern-list ')'

pattern-list
  -> pattern/','

pattern
  -> positional-pattern

positional-pattern
  -> inductive-pattern
  -> binding-pattern
  -> zero-pattern

inductive-pattern
  -> pattern-decl subpatterns

binding-pattern
  -> id

zero-pattern
  -> '0'

pattern-decl
  -> pattern-spec id?

pattern-spec
  -> id alpha-type-params?

subpatterns
  -> '(' subpattern-list? ')'

subpattern-list
  -> subpattern/','

subpattern
  -> positional-pattern
  -> nonpositional-pattern

nonpositional-pattern
  -> explicit-pattern
  -> implicit-pattern
  -> ellipsis-pattern

explicit-pattern
  -> id ':' positional-pattern

implicit-pattern
  -> id ':'

ellipsis-pattern
  -> '...'

guard
  -> '|' '(' balanced-item+ ')'
```

```
balanced-item
-> balanced-item-no-comma
-> ','
```

```
balanced-item-no-comma
-> alpha-nested-item
-> '(' balanced-item* ')'
```

```
-> '[' balanced-item* ']'
```

```
-> '{' balanced-item* '}'
```

```
-> any C++ token but ',' not conflicting with any of the above
```

```
balanced-items-list
-> balanced-items-no-comma/','
```

```
balanced-items-no-comma
-> balanced-item-no-comma+
```

Notes.

An "id" is any valid C++ identifier.

A "string-literal" is any valid C++ non-multi-part string literal e.g. "xy" is okay but not "x" "y".

[CONTENTSNEXT](#)

Glossary

algebraic (data) type

A sum of products (akin to a discriminated union).

alpha

An app specific synonym for "algebraic" (more or less).

alpha base type

A relatively simple kind of algebraic type (the alpha typing "base case").

alpha data type

The app equivalent of a general algebraic type.

alpha forward type

A mutually recursive alpha type.

alpha type

An alpha base type, unit type, data type, or type type.

alpha type type

Like a mutually recursive typedef.

alpha unit type

A singleton sum nonrecursive alpha data type.

binding pattern

A pattern that corresponds with a C++ variable (unless it is a wildcard).

concrete data type

Behaves like a native type; see Coplien, "Advanced C++".

handle/body idiom

An interface/implementation separation device; see Coplien, "Advanced C++".

inductive pattern

A pattern that may have nested subpatterns.

injector (itor)

A generated class corresponding to a sum component.

null (zero) alpha object

A handle object with a zero body object reference (null pointer).

nullary ctor

A handle class default ctor that instantiates a body class object.

parametric alpha type

Has type parameters that translate to template parameters.

product (component)

A conjoint constituent of a sum component.

projector (ptor)

A generated product access declaration.

root class

A generated class corresponding to an alpha data type.

selector (stor)

A generated sum discrimination expression.

subject

An alpha-typed C++ expression as a match statement subjects element.

sum (component)

A disjoint constituent of an algebraic type.

unary ctor

A ctor that can accept a single argument when applied.

wildcard

A binding pattern of the form '_'.

zero pattern

A pattern of the form '0' for matching null subjects.

[CONTENTSNEXT](#)

Changes

App 2.2 [released 29 May 2000]

- Certain beta.h functions have been rewritten to avoid requiring default ctors of their type parameters. With the possible exception of List<A>::at, which now always escapes for invalid indices (the optional 'escapes' parameter has been eliminated), the changes should be transparent. The changes affect all functions that previously required default ctors, with the exception of Associative_List<A, B>::get, which remains unchanged.
 - New files shared.h & _shared.h have been added to applib with classes that provide general non-intrusive reference counting capabilities. The shared.h file is an app source; _shared.h is the app generated version.
 - A new core function escape_output has been added to set/get the output stream used by escape for when APP_NO_FLOW is defined; the Flow code now uses a current output stream defined by a new function Flow::output.
 - The 'Extending_List' beta.h class has been augmented with new functions that give it stack/queue/deque capabilities.
 - The app syntax now allows the 'explicit' C++ keyword for base types, unit types, and data type itors. The keyword applies to the associated handle class ctors that instantiate body objects.
 - A new option '--stor_method=m' has been added, where m=d means use (as usual) dynamic_cast in generated runtime type selection code, and m=r means use a new alpha core function "raw_type_is" in generated code, which provides for generally faster (but perhaps less generally applicable) generated code. The default remains dynamic_cast. The option may be applied either globally via the command-line, or locally on a match statement basis via the \$option syntactic construct.
 - Option '--skeletons=f' is now available via the \$option construct.
 - The Match and Alpha_Type_Decl Local_Formatter classes have been defriended, by publically exposing relevant member variables via accessor functions, allowing for easier customization of the internal Local_Formatter classes.
 - Added defines in applib_config.h to make the core allocation and deallocation functions easier to customize and more efficient via conditional inlining.
 - Top_::report now has the synonym Top::report (for convenience).
 - Fixed bugs:
 - ◆ A match statement embedded within the user-code part of an alpha type declaration (e.g. in the body of an inline member function) is translated to the initial output stream rather than the string stream associated with the related filled skeleton.
 - ◆ In beta.h the == and < operators for Single types contain typos that don't show up as errors unless the operators are actually used. As a workaround, if needed, edit (and re-app) the beta.h file so that "empty" replaces "no" and "not_empty" replaces "yes".
 - Known unfixed bugs:
 - ◆ (none)
-

App 2.1 [released 26 Feb 2000]

- App now runs under g++ 2.95.2 (cygwin) and 2.91.66 (slackware 7 linux). The g++ support for iostreams is however still a little buggy, and as a result there are certain limitations: option '-ori' is ineffectual, and #line directives that point back to the generated output file are not generated (the app code specializes the filebuf and ostream classes and that appears to be a problem with g++). Also

g++ is pretty sensitive to namespace issues and refuses to compile the test.cpp app source file, so that code is stubbed out under g++ (an internal issue that doesn't impact users directly).

- The installation scripts now reapply app to beta.h prior to copying to the INCLUDE location if needed, to adjust generated #line directives in _beta.h to reflect an INCLUDE differing from the default INCLUDE.
- The native type 'float' and boxed version 'Float' have been added to beta.h as alpha base type declarations.
- Option '-o=-' ('--output=-') now means: use cout for output.
- The applib_config.h file now conditionally defines 'alpha' as APP_LIB_NS a.k.a. 'applib' (by default).
- The Top__::report function now returns the byte leak count as a result and also has a new parameter 'quiet' (default false) to tell it not to write.
- Fixed bugs:
 - ◆ In beta.h the List<A> concat(A) & extend(A) functions should be defined outline not inline (due to the forward Cons<A> itor references). This would cause compile-time errors but only when you actually tried to use them (with MS VC++; with g++ the errors are reported in any case). As a workaround, edit (and re-app) the beta.h file so that those functions are defined outline ala List<A>::length() etc.
 - ◆ In beta.h the base type declarations for Associative_List & Extending_List are missing template parameters. This would cause app warnings or compile-time errors but only when you actually tried to use them. As a workaround, edit (and re-app) the beta.h file so that:

```
$base Associative_List<class A, class B>;
$base Extending_List<class A>;
```

- Known unfixed bugs:

- ◆ (none)

App 2.0 [released 22 Jan 2000]

- Nonpositional (a.k.a. keyword) pattern matching is now supported; a pattern of the form "x:y" is nonpositional, where x is an alpha unit or data type product-decl id and y is a positional pattern that may contain nonpositional subpatterns. A nonpositional pattern of the form "x:" is taken as shorthand for the nonpositional pattern "x:x". A nonpositional pattern of the form "..." is taken as "any other pattern not explicitly mentioned is '_' i.e. a wildcard". App internally rewrites nonpositional patterns as equivalent positional patterns. Here's an example.

```
$data A = B(int c, Atom d, A e) | F();
...
A a = B(0, "ok", B(0, "ok", F()));
...
$match (a) {
  (B(e:B(d:g, c:, ...), ...)) => {...c...g...} // internally rewritten as:
  (B(_, _, B(c, g, _))) => {...c...g...}
  ...
}
```

- A concept called a "safety case" has been introduced. A safety case is any pattern case that follows another pattern case in which all patterns are ids including '_' (i.e. any pattern case that follows a default case). Safety cases are useful when the efficiency implied by default cases is desired but for case coverage analysis and documentation purposes it is also desired to explicate all relevant cases. In effect they prevent default cases from short-circuiting app's analysis algorithm. Pattern cases

following a default case are now treated as irrelevant at run time and are excluded from the translation. App warns you if a safety case has associated guard or statement code. Here's an example.

```

$data A = B(int c, Atom d) | E(int f) | G(Atom h, int i);
...
A a = B(0, "ok");
...
$match (a) {
  (B(x, y)) => {...}
  (_) => {} // default case
  (E(...)) => {} // safety case
  (G(...)) => {} // safety case
}

```

- If a default case has no associated statement code it is now excluded from the translation (default cases do not have associated guard code).
- The generation of #line directives has been improved. Directives are now generated to reset the location at the end of a nested match statement when it appears as a final balanced-item* in a balanced-item-no-comma construct. Directives are also generated to account for guard code.
- A new option '--only_read_includes' causes app to treat the argument files f1..fn-1 as implicit include files as usual but which are excluded from the generated code. By using explicit #include directives for _f1.._fn-1 in your code together with this option (assuming default _f1.._fn-1 app outputs), you can reduce the size of the generated code and also the number of redundant symbols. This is particularly effective for use with class browsers etc.
- All generated classes are now user-extensible/definable via the \$user {...} construct, including itor handle classes and forward root classes.
- A static function nil() has been added to the common handle class methods for default skeletons so that generated handle classes provide an alternate way to generate corresponding null objects via function nil().
- The restriction that outer binding patterns (ids) must be wild has been removed. Though such patterns are unusual there is no real reason to prohibit them; the restriction simplified the implementation slightly but otherwise was artificial.
- Casting declarations are now not generated for inductive patterns that have no subpatterns and no optional id in the associated pattern-decl construct, a minor optimization (also a bug fix - q.v.).
- Pretty print indentation for generated "void write(ostream& _os_)" functions now depends on option line_indent_delta instead of being fixed at 2. The generated write() functions now rely on Alpha_Control functions to handle markup formatting, providing more customization capabilities. Additionally the alpha.h write_string() function is now customizable via Alpha_Control, and the Alpha_Control interface has been extended with a few new functions.
- The applib Alpha_Control and Flow classes can now be customized at any time.
- The Copy type in alpha.h is now a class, and associated enumerals nocopy and copyok are now const variables (this should be a transparent change).
- The Str class and Atom applib alpha types can now share dynamic buffers via ownership transfer.
- Added a new applib class Pretty_Printer that customizes the Alpha_Control alpha.h core class, providing pretty printing capabilities for alpha types.
- Augmented the grammar for base type declarations so that e.g. "\$base int;" is taken as equivalent to "\$base int() \$user {} \$user {};".
- The beta.h library code has been augmented; changes are upwards-compatible:-- added declarations for certain "unboxed" native types e.g. bool, char, etc.; added a new Associative_List ctor that accepts an initial state argument and also new add/addnew overloads; added a new class Extending_List for efficient repeated destructive list appends; Associative_List now appears as an alpha base type as does Extending_List; added a new alpha type Single that models singleton sets; added const function qualifiers to the first() etc. Pair and Triple access functions. Added bool

Top::not_null() and char* new_string() to alpha.h, and augmented dup_string() with an optional parameter to control use of ::malloc v. new[].

- The app source has been generally improved:– the generated code for match statements is a little more optimal; a new factory class allows for easier customization at a finer (class) level of granularity; the messaging code is easier to customize (via the factory class); the symbol table code uses more explicit alpha typing together with nonpositional patterns.
 - The "docin" code has been replaced by a simpler version, "docins".
 - Fixed bugs:
 - ◆ If a pattern–decl construct has an optional id but there are no associated subpatterns, as for example in the pattern "Foo f()", the corresponding generated casting declaration is omitted if the declarations for the other patterns in the associated pattern case are also omitted. This would cause a compile–time error. As a workaround, the id can be omitted, eliminating the need for a (missing) casting declaration, if the corresponding subject expression is named – if not already – and that name used instead.
 - ◆ Certain generated #line directives relating to nested inductive patterns are of the form #line 0 "" and could cause confused compiler messages. As a workaround, option '-nolds' can be used if/when there is a problem.
 - ◆ If the patterns in an initial pattern case are all ids including '_', the generated conditional expression is empty and would cause a compile–time error. This situation is unlikely as such cases effectively trivialize match statements.
 - ◆ A guarded pattern case in which all patterns are ids including '_' is considered a default case, and could cause omission of generated match statement epilog code that calls the escape() alpha.h core function if matching fails. This situation is unlikely as such pattern cases make little sense.
 - ◆ If a "=> const" accessor construct appears, the return type of generated access functions is not const qualified.
 - ◆ The List<A> comparison operators >= < > <= in beta.h have typos.
 - ◆ Empty guard expressions are accepted and which cause compile–time errors, a syntactic technicality.
 - ◆ Empty inputs are not accepted, contrary to the grammar.
 - Known unfixed bugs:
 - ◆ (none)
-

App 1.0 [released 22 Sep 1999]

- Initial release.