Reprinted from *ACM SIGPLAN Notices* 35(3):73–83, March 2000.

The 1999 ICFP Programming Contest

Norman Ramsey
Harvard University
nr@eecs.harvard.edu

Kevin Scott
University of Virginia
jks6b@cs.virginia.edu

Introduction

The International Conference on Functional Programming (ICFP) sponsors an annual programming contest. The most recent contest was held from September 2 to September 5, 1999. The contest is primarily for fun, but it also serves two serious purposes: to show how functional languages can help people solve hard problems quickly, and to provide a "reality check" by comparing programs written in functional languages to programs written using the more popular imperative and object-oriented paradigms. The contest is open to all, and entrants may use any programming language—the only limitation is that entries must work on a Pentium running Linux. The Judges award First and Second prizes based on objective criteria, and they award a Judges' Prize to another entry which, in their subjective opinion, is most deserving. The prizes include cash awards, books on functional programming, and peer recognition, including "unlimited bragging rights."

Entrants may enter alone, or in teams of any size. They are given 3 days (72 hours) to solve a programming problem. There is also a special "lightning division," consisting of entries submitted within 24 hours. Entrants in the lightning division compete on an equal basis with other entrants, but they get more notoriety if they win—and they get to spend more time with their families.

Many programming contests set problems that, at least in principle, can be completely solved in the time allotted. The ICFP contest uses more challenging problems. We have tried to choose problems that permit most teams to come up with some reasonable solution in 24 hours, but that have depths or ramifications that cannot be fully explored even in 72 hours. The 1999 problem involved two subproblems: generating good code for the case statement, which has well known solutions, and compiling pattern matches, which is widely believed to be intractable. The contest's 37 entries included many worthy solutions, including a first-place winner whose performance can only be described as dazzling.

The 1999 contest was graciously supported by ICFP and ACM, who donated cash, and the MIT Press, Cambridge University press, and Michel Mauny, who donated books.

The problem

The ICFP contest draws entrants with widely varying abilities—from high-school students to the world's top professionals. There are no perfect problems for such a diverse crowd, but we hope a good problem has these characteristics:

- It admits of many solutions of varying quality—there should be no clear "best" solution.
- As noted above, the problem is easy enough that one can write a simple solution in 24 hours, but hard enough that 72 hours is not long enough to explore all its ramifications.
- The problem should reward algorithmic cleverness more than raw computing power.
- It should be possible to evaluate up to 50 entries in just a few weeks.

We crafted a contest problem that combined two well-known compilation problems: generating good code for the case statement, and compiling pattern matching to efficient decision trees. To avoid making it immediately obvious what the problem was, and to make the problem more attractive to people not obsessed with compilers, we added some window dressing regarding interactive fiction on handheld computers.

Interactive fiction, sometimes known as "text adventure gaming," typically involves human players interacting with a virtual world containing locations, objects, and other "non-player" characters. During the 1980s, the heyday of interactive fiction, interest in the genre rose and fell with the fortunes of Infocom, purveyors of *Zork* and other amusements. Interest grew

Bridge Street
A busking magician stands behind a folding trestle table, exhorting passers-by to come and watch his tricks. On the table are three inverted cups (one red, one green and one blue).

"Roll up! Roll up!" calls the busker.

>take green cup
"I don't mind if you look under the cups," says the busker, "but I'd be grateful if you didn't walk away with them."

>look under blue cup
There's nothing under the cup.

"Who'll come and see my magic?" calls the busker.

>wait
"Three perfectly ordinary coloured cups," says the busker. He lifts each in turn to show that it is empty. "Nothing under any of them," he says.

⋮

Figure 1: An interaction with a non-player character

behavior	⇒	({*rule*})
rule	⇒	({*state-num*} *stmt*)
stmt	⇒	(`IF` *condition stmt* ({*elseif*}) *stmt*)
	|	(`DECISION` *new-state utterance*)
	|	(`CASE` *variable* ({*arm*}) *stmt*)
elseif	⇒	(`ELSEIF` *condition stmt*)
arm	⇒	(`ARM` *value-set stmt*)
condition	⇒	(`EQUALS` *variable int*)
	|	(`AND` {*condition*})
	|	(`OR` {*condition*})
variable	⇒	(`VAR` string)
state-num	⇒	*int*
new-state	⇒	*int*
	|	*wildcard*
wildcard	⇒	_
utterance	⇒	*string*
value-set	⇒	({*int*})

A phrase enclosed in curly braces ({...}) is to be repeated zero or more times. An *int* is an integer literal and a *string* is a string literal. Valid behaviors satisfy these properties:

- In each behavior, each *state-num* labels exactly one rule.
- In each `CASE` statement, each *int* appears in the *value-set* of at most one arm.

Access to the current state is available through the state variable called `state`.[2]

Figure 2: The language to be optimized.

again during the 1990s, largely owing to the availability of programming languages designed especially for interactive fiction, among which the most notable are Inform (Nelson 2000) and TADS (Roberts 1996). Although these languages do an excellent job helping authors manage locations and objects, creating interesting non-player characters is still painful. Figure 1 illustrates a fragment of an interaction with a non-player character from Gareth Rees's (1995) popular game, *Christminster*. Programming such a character in Inform or TADS is little better than writing C code. The premise of the contest was that Frobozz Role Playing has designed a new language especially for specifying the behaviors of non-player characters. The contest problem was to compile these behaviors into code for wireless, handheld devices.[1] For simplicity, we imagined that non-player characters were compiled to state machines.

> *The "mimesis team" has developed a new, ultra-high-level language to specify the behaviors of the computer-controlled characters. Their high-level compiler translates these behaviors into state machines. Unfortunately, the logic used to decide on state transitions is looking very complicated, and management worries that the state machines may exceed both code-size and power budgets for the hand-held devices—and nobody is going to buy unless price is low and battery life is high. Your task, for the next 72 hours, is to write an optimizer that will improve the state machines emitted by the compiler. Speed is nice, but management worries more about code size—if the system is 10% slower than projected, users might not notice, but if the system is 10% larger than projected, that means more memory, and there go the profits.*

Figure 2 specifies the language in which state machines are expressed. When an event occurs, the system finds the rule for the current state, and it evalutes the rule to compute a new state for the character, as well as an "utterance" (which might presumably be printed to the screen). The rule may use both `CASE` and `IF-THEN-ELSE` expressions to decide on a new state and an utterance. Entrants were warned that test inputs were unlikely to make effective use of `CASE`. The full problem statement, which may be viewed at `http://www.eecs.harvard.edu/~nr/icfp/problem.html`, also included *rewrite rules*, which specified the exact semantics and cost of evaluation, and a *size function*, which gave the expected code size of a character. To derive the cost estimates for both time and space, we devised an idealized RISC instruction set and wrote a translator that

[1] We all need to play on the subway.

Entries	*Language*
9	C
6	Haskell
6	OCaml or OLabl
5	C++
2	Scheme
2	Python
2	ML (1 MLj, 1 Moscow ML)
2	Perl (1 using first-class functions!)
1	Prolog
1	TALx86 (a typed assembly language)
1	Java
1	Icon

Table 1: Languages used in the contest

emitted those instructions.

In sum, the contest problem was to take an arbitrary behavioral description written in the language of Figure 2, and to emit an equivalent description with a code size as small as possible.

The Entrants

37 teams from 10 countries entered the contest.[3] Entries came from Australia, Canada, France, Germany, Italy, Japan, the Netherlands, Sweden, the United Kingdom, and the United States. Teams ranged in size from 1 to 8 people, and included high-school students, college students, graduates, faculty, and people with real jobs. Of the 37 entries, 6 were "lightning" entries completed in 24 hours or less. Over a third of the entries were in C or C++; among other languages, the lazy functional language Haskell and the strict functional language Objective Caml were the favorites. Table 1 shows the programming languages used in all the entries.

The Judges evaluated the entries in two phases. The preliminary rounds produced 8 finalists, whose identities were announced before the conference. The final round determined the winners.

The Preliminary Rounds

In the preliminary rounds, entries were evaluated according to three criteria:

- Ability to produce *syntactically* correct descriptions.
- Ability to produce *semantically* correct descriptions.
- *Code size* of the descriptions.

Regrettably, the "mimesis team" and their "ultra-high-level compiler" existed only in the minds of the Judges. The test inputs were therefore drawn from four sets:

- *Hand-written descriptions.* These inputs are straightforward hand translations from Inform source code, meant to capture at least the spirit of four characters from Christminster: the the busker, the cat (Turmeric), Dr. Jarboe, and the parrot. One of these descriptions, the busker, was made available during the contest for use as a test case.
- *Machine-lowered non-player characters.* The hand-written descriptions are written at a fairly high level, but we felt that a high-level compiler would be likely to produce descriptions at a lower level than what we wrote by hand. We therefore ran an artificial "lowering" pass, which replaced `CASE` statements with `IF` statements.
- *Machine-generated non-player characters.* These fiendish test cases were drawn from another problem domain entirely: recognizing machine instructions.

Round Zero: syntactic correctness Surprisingly, at least a quarter of the entries had difficulty even producing a syntactically correct behavior for a simple input. Difficulties included:

- Dumping core on every input, because of configuration problems.
- Taking the string on the command line as a file name, not a time limit.
- Failing to wrap parentheses around one part of the syntax or another.
- Emitting `nil` instead of `()`.

Most of these problems were repaired either by the entrants themselves or by the Judges; we tried not to penalize any entry for stupid syntactic slips. The most interesting problem repaired by the Judges was that one entry nested Objective Caml's `Format` boxes too deeply, and we had to add `Format.set_max_boxes 1000` in order to get its output.

After effecting these repairs, we eliminated those teams not capable of producing syntactically correct output for each of the simple hand-written non-player characters. The upper half of Table 2 shows these entries. Unsafe languages predominate, although two of the incorrect entries were written in the safe language Perl. None of the failing entries were lightning entries.

[2]The ability to get to the state through a named variable was added partway through the contest, when it became clear that otherwise it was going to be hard to produce interesting optimizations. Had we realized this earlier, we would have chosen to represent each behavior by a simple `CASE` statement.

[3]A 38th entry was summarily disqualified for mounting an attack against the Judges' machine.

Team	*Language*	*Failure*
Team Squirrel	Perl	Irreperable confusion about reserved words
Team Sasquatch	C	Errors in parsing
Ignition	C	Dumped core on parrot
team mutato	Perl	Deep recursions, uninitialized variables, missing *elseifs*
Pizza Bison	C	parser errors
The lone coders	C++	Assertion failures
Seth Pensack-Rinehart	C	Segfaults in `parse()`
Lone Wolf	C	Wrong answer on a recognizer for SPARC addressing modes
Nederwiet	Haskell	Wrong answers on lowered non-player characters
C++ Paladins	C++	Wrong answers on parrot
CyberTiggyr	C	Wrong answers on cat, parrot with `IF` for `CASE`
Ulm Sparrows	Haskell	Wrong answers on cat
Wolfram Kahl	OLabl	Wrong answers on cat
Randy, Marianne & the Cats	C	Wrong answers on cat, parrot
jkg	C	Wrong answers on a recognizer for the SPARC instruction set
Jérôme	OCaml	Wrong answers on the "pessimized" parrot

Table 2: Teams eliminated in the preliminary rounds

Round One: semantic correctness In this round, the Judges figured anything was fair game. An optimizer should produce correct output for *any* correct input, no matter how different that input may be from what the optimizer expects. Accordingly, the Judges used all available descriptions to test the remaining entrants for semantic correctness.

Testing for correctness means comparing the behaviors of two descriptions in the same initial state. Finding a representative set of initial states is nontrivial. We chose states as follows:

- For each variable in each non-player character, compute the set of values against which that variable is compared (either by `CASE` or by `EQUALS`). Then find an integer not in the set, and add it to the set.

- To get the set of initial states, take the Cartesian product of all the sets.

The Judges claim that such a set of initial states is guaranteed to exercise all reachable paths in both unoptimized and optimized programs. For some non-player characters, like the busker and Dr. Jarboe, which have test sets of 1,872 and 768 states respectively, the Judges tested non-player characters exhaustively. But this was not possible for all non-player characters; the parrot has a test set of 3,611,520 states, and some of the machine-generated non-player characters have test sets many orders of magnitude larger. The largest test set has on the order of 10^{400} states. This is an interesting number to work with; not even a 64-bit IEEE floating-point number can represent it.

The Judges ran exhaustive tests on descriptions with up to 40,000 states; on larger descriptions, we tested 10,000 states selected at random without replacement. This round eliminated more entries than the preceding round, as shown in the bottom half of Table 2. The most common source of a wrong answer was incorrectly replacing a state number with a wildcard in a `DECISION`. The Judges were especially sorry to have had to eliminate the last two entries, because both of them were highly effective on many descriptions—each produced wrong answers on just a single description.

Again, none of the lightning entries were eliminated. The entries eliminated for semantic errors include not only programs written in C and C++, but also programs written in the statically typed functional languages Haskell, OCaml, and OLabl. Evidently, type-checking does not always guarantee correctness.

Round two: size of hand-written non-player characters after optimization The hand-written non-player characters are the most like what we told the entrants to expect, so we decided that performance on the hand-written non-player characters was the best criterion by which to decide who should advance to the finals. We therefore tested the entries on the four hand-written non-player characters, with the results shown in Table 3. The rightmost column shows the size of the optimized code divided by the sized of the unoptimized code. Smaller percentages are better; the top 3 teams better than halved the sizes of their descriptions. In Table 3, "lightning" entries are marked with *. All six of these entries were finished in under 24 hours—and all six are correct. Three of the six perform significant optimizations, and indeed two of those sit squarely atop the standings. Five of the six use functional languages. We think these results speak for themselves.

Team	*Language*	*Size ratio*
⇒LA*	Haskell	48.3%
⇒Si³*	Haskell	48.9%
⇒Camls 'R Us	OCaml	49.4%
⇒Indiana University	Scheme	55.3%
⇒KABA	OLabl	57.0%
⇒Bat8*	OCaml	57.7%
RST HuckerMucks	C++	65.6%
Hammond & Sampson	C	75.7%
rob	C	84.0%
⇒Two Guys	Java	92.9%
Unofficial OK State	Icon	94.3%
U-Penn PL Club	Scheme	94.4%
⇒Jeremy Dawson	Moscow ML	95.9%
PandP	Perl	96.0%
UNSW PHANTOMS	Haskell	96.2%
NoTime	Python	98.3%
Ulm Sparrows*	Haskell	98.5%
Bruce*	ML-J	100.0%
4:am Productions	Python	100.0%
danknamorrisq	C	134.9%
Team TAL*	TALx86	7,392,370

Teams marked with a * were "lightning" entrants.
Teams marked with a ⇒ continued to the final round.
Size ratio is final code size divided by initial code size.

Table 3: Team rankings after the preliminary rounds.

After the preliminary rounds, the top six teams advanced to the finals. The Judges also chose to advance two teams that did not finish in the top six overall, but that got top scores on at least one of the four hand-written non-player characters. In Table 3, finalists are marked with ⇒.

The winners

As at the 1998 contest, there was one entry that, although not deemed worthy of the Judges' Prize, was still deemed worthy of recognition. Readers may have guessed from Table 3 that there was something unusual about the Team TAL entry. Here is an excerpt from the description of that entry, the full text of which may be found at `http://www.cs.cornell.edu/talc/icfp99-contest/contest.htm`.

Our submission is in TALx86, a strongly typed functional language that encourages an explicit continuation-passing style and supports mutually recursive modules. We were encouraged to use this language when we learned that the competition would allow us to run our program on an interpreter implemented in hardware. We are grateful to the Intel Corporation for developing this interpreter.

We introduce the terms and primitive constructors of the language below, but programmers experienced in using functional languages may prefer to jump right into the code we have provided as a solution to programming contest. We believe the explicit type annotations on distinguished TALx86 continuations make the code essentially self-documenting. [An excerpt appears in Figure 3; `LABELTYPE` marks one such annotation.]

A TAL module is an ordered list of declarations followed by an ordered record of continuations followed by an ordered record of primitive values. In the concrete syntax, these components, or "segments," are delineated by the reserved words CODE and DATA. We focus on the ordered record of continuations since it is the most useful for the TALx86 programmer.

The *simplest* way to understand these continuations is to view them as syntactic sugar for a strongly-typed, closed, continuation-passing lambda calculus augmented with a number of primitive operations. TALx86 also provides convenient support for traditional imperative operations by encouraging a store-passing style and by providing syntactic support for composing store-passing continuations. In fact, much of the power and expressiveness of our system revolves around the continuation-composition operator.

In the concrete syntax, the continuation-composition operator is the newline character. Given two continuations c_1 and c_2, the composition operator creates a new continuation c_3 with arguments s and k. In familiar terms, c_3 may be thought of as

$$\lambda sk.c_1 s(\lambda s'.(c_2 s'k)).$$

The type of a continuation is expressed by stating the types of state and continuations which it may consume. The most important part of a state type is the "register file type" which is a record type with 8 fields. (Programmers often express frustration that this type does not admit wider records, but Intel has remained unresponsive.)

We have chosen TALx86 as our source language because it contains primitives at an appropriate level of abstraction for completing this task. In particular, its vast collection of arithmetic operations and the efficient mechanisms for composing computations give our team the edge we need when it comes to computing the complicated cost metrics that we have determined constitute the bottleneck. Although some observers may claim that the TALx86 primitives are somewhat low-level, we defend our choice by pointing out that the primitives are asymmetric enough to allow a very compact encoding, especially if you use the TALx86 binary format.

Our solution exploits well-known properties of the plus operator on the Pentium architecture. Since the contest specified that our program would be run on a Pentium, we assumed that the program to test the size of our output would also be run on a Pentium. Therefore, the reasonable semantics to ascribe to the + operator in the definition of program size is 32-bit addition on a Pentium

```
; TAL IMPLEMENTATION compute
        INCLUDE TAL.INC
        _begin_TAL
        TAL_IMPORT      pop_runtime.tali
         .
         .
         .
        TYPE    <Ast?stmtA =^+[*[S(3)^r,(^*[str^rw,(‘List?list? ^*[(‘List?list? B4)^rw,‘Ast?stmt?^rw])^rw,
          ‘Ast?stmt?^rw])^r],*[S(2)^r,(^*[(‘Core?Opt? B4)^rw,str^rw])^r],*[S(1)^r,(^*[(‘List?list?
          ^*[‘Ast?condition?^rw,‘Ast?stmt?^rw])^rw,‘Ast?stmt?^rw])^r]]>
        CODE
_pop_main:
LABELTYPE <All[s1:Ts s2:Ts e1:Tcap e2:Tcap].code{cap: &[e1,e2],EBP:sptr (H e2 s2)::s2,ESP:sptr (code{cap: &[e1,e2],
          ESP:sptr s1@(H e2 s2)::s2,EBP:sptr (H e2 s2)::s2})::s1@(H e2 s2)::s2}>
        SUB     ESP,12
        MOV     DWORD PTR [ESP+4],EBP
pop_main$2:
        ADD     ESP,4
         .
         .
         .
```

Figure 3: Excerpt from the *Team TAL* entry
(Lines containing `TYPE` and `LABELTYPE` have been wrapped.)

processor. Therefore, the smallest programs are those with a size that is a multiple of 2^{32}. (Since size normally connotes the impossibility of negative values, the sizes we compute should be interpreted as unsigned 32-bit numbers.)

To create a program with size a multiple of 2^{32}, we exploit the size of the case statement, in particular the span of the arms. We simply replace the body of the first rule in the program (call it `s`), with the following equivalent one (where x is a meta-variable explained below):

```
(CASE (VAR "state") ((ARM -2147483648 s)
                     (ARM x s))
                     s)
```

We note that for any int x, this statement is well-formed and equivalent to `s`. It simply remains to identify an optimal value for x. Our program first computes the size of the whole program (call it $size_P$) and the size of s (call it $size_S$). We would like an x such that

$$2^{32} = size_P + 10 + size_S + size_S + (x + 2^{31} + 1).$$

Hence,

$$x = 2147483647 - size_P - 10 - size_S - size_S$$

where the symbol "$-$" designates subtraction mod 2^{32} (see Intel Architecture Software Developer's manual, Volume 2, pages 3-448 and 3-449 under the title "SUB" for a more precise specification).

Unfortunately for Team TAL, the contest entries were evaluated not by a C program, but by a program written in Standard ML, which detects integer overflow. After the evaluation program was rewritten to use arbitrary-precision arithmetic, Team TAL did indeed produce "optimized" descriptions of size 2^{32}, explaining their dead-last placement in Table 3. But the Judges felt that such a virtuoso performance had to be recognized, and indeed, that there was only one possible way to do so. Dan Grossman, Fred Smith, Dave Walker, Stephanie Weirich, and Steve Zdancewic were presented with a copy of *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, by Donald E. Knuth. Readers who are curious about TALx86 can learn more at `http://www.cs.cornell.edu/talc`.

The Judges' Prize The Judges were surprised by the variance in the quality of the entries. Many entries failed to work or to produce correct output, but there were also many well done, clever entries, particularly among the finalists. Even in this very strong group, however, one entry stood out.

- It was a lightning entry, completed in under 24 hours.
- It was written in Haskell, totalling only 376 lines of code. By comparison, the other finalists averaged over 1500 lines of code.
- The optimizer itself is only 137 lines. If we remove blank lines, comments, and module `import` statements, it is exactly 100 lines.
- The entry topped the preliminary standings.

Here is the author's description of the entry from Team LA.

It was Thursday evening, 11:00 PM. I was watching TV with my wife when I realized that the ICFP'99 programming contest had just begun. I went over to our office at home and studied the task. After cutting through some nonsense about interactive fiction I finally found out what to do.

After looking at the sample file, I could spot some possible optimizations. The task at hand was ideal for functional languages and I chose my favorite, Haskell. You obviously needed a parser and a printer for the optimizer. I used a parsing combinator library for the parser. While coding it I mumbled something about how biased this was in favor of Scheme. They should be able to use the built in parser and printer. A couple of hours later the parser and printer worked. Combining these, I had written a complicated identity function, and satisfied I went to bed.

The next day I discussed the problem with some colleagues of mine; we concluded that the right way to solve the problem was to generate the exact condition for `DECISION`, put these into some normal form, and crunch away with optimization on that. This strategy would eliminate any bias due to the way the input was coded. But I decided I didn't have time for such a fancy solution. Instead, I used the good old compiler writer's trick of staring at the compiler output and figuring out some simple local optimization that would improve it. This what the program ended up as, just a bunch of simple local optimizations.

- While optimizing a CASE arm, keep track of the value of the scrutinized variable to avoid testing it again.
- Join CASE arms that contain very similar CASE statements.
- Remove IF with identical branches.
- Avoid the number in a DECISION if possible.
- If the gap between elements in a CASE range is larger than the expected cost of a new CASE or IF, split the CASE range.
- Treat the top level of the program as a CASE internally to simplify the code.

That's it. Embarrassingly simple. I probably spent about 6 or 8 hours on the task. The program contains about 300 lines of code, so I guess my productivity wasn't that high.

The Judges disagreed with this conclusion so completely as to award this entry the Judges' Prize, which includes the statement that, in our professional judgment,

Lennart Augustsson is an extremely cool hacker.

Readers who are curious about Haskell can learn more at `http://www.haskell.org`.

The Final Round Because competition was so keen among the top entries, the final round used more demanding tests than those used in the preliminary rounds:

- Four character descriptions, with representations "lowered" to a long list of $\{condition \rightarrow action\}$ rules.
- Four recognizers for hardware instruction sets, automatically translated into the contest format.

By totalling code sizes before and after optimization, the preliminary rounds gave the most weight to performance on the easiest test cases, i.e., the ones on which there was most room for improvement. Because the final test cases included some that were fiendishly difficult, in the final round we used a system that weighted each test case equally. We used each test case to rank the entries, then combined the rankings using an Australian ballot. To win the contest, it wouldn't be enough just to crush the easy cases; the winner would have to do well on a diverse set of cases.

Second Place The second-place winner was also a lightning entry. Although considerably longer than Lennart Augusston's prizewinning entry, it was still small for a finalist, at only 1250 lines of Haskell. As with the other two top entries, this total not only included an optimizer, but also code to generate test data and interpret the optimized description, suppressing optimization if it found an inconsistency. To earn second place overall, Team Si^3 took second place on 5 ballots, with these ratios:

54.3% 95.2% 90.4% 80.6% 92.1% 89.8% 92.0% timeout

Again, an excerpt from the authors' description.

The problem specification was released at 10pm local time. One of us spent three hours reading and thinking before going to bed. (The other two were in the pub.) We set to work in earnest at 9am the following day, and delivered our final program 13 hours later, just inside the 24-hour time limit.

Fortunately, the problem decomposed easily into three parts:

- A generic simplifier.
- A branch-merging engine.
- A parser, pretty printer, driver, test harness, and test cases.

We coordinated our work using a CVS repository, which was hugely useful in making sure we could work together without tripping over each other, or losing patches in a blizzard of changes.

Our optimizer converts IF to CASE, then uses these passes:

- **Simplify**. The simplifier performs these transformations:

- Merge identical branches of IF/CASE
- Eliminate IF/CASE with only one remaining branch
- When enclosing cases provide information about variables,
  * prune case branches that cannot match, and
  * eliminate CASEs that test the value of a known variable
- Simplify Boolean conditions arising from the above
- Replace cascaded CASEs (where the DEFAULT branch scrutinizes the same variable as the outer CASE) with a single CASE.
- Replace "DECISION n" with "DECISION _", using the cheaper wild-card form, where possible.

The simplifier carries an environment that maps each state variable to the information known about it at that point in the code. In particular, in a CASE alternative we know that the variable scrutinized by the CASE has a value that matches the pattern.

- **Reorder**. The program is basically a decision tree. The reorder pass tries to change the order in which variables are examined in order to produce a smaller tree. The algorithm is simple and brutal. For each top-level expression, we examine all the paths from the root to a leaf of the tree. We identify variables that are scrutinized by a CASE on a high proportion of these paths. For each such variable we try the effect of building a new outer CASE that scrutinizes the variable and has arms corresponding to the values with which it was compared on the paths we computed earlier.

  Next, we simplify the resulting CASE, using the same generic simplifier; the simplifier takes account of the information from the new outer CASE to eliminate inner CASES that scrutinize the same variable.

  Finally we compare the size with the expression we started, and accept the change if it has shrunk.

- **Merge**. For each case, try to merge branches that are "fairly similar." The differences are patched up by adding cases at the patch points. This pass made a huge difference for some programs (e.g., the Busker).

  Merging is tried for each CASE in two stages. First, we try to merge *all* the branches into a single branch. If that doesn't work, we try merging branches on a pairwise basis, checking for each pair whether merging would make the code smaller or not. The first stage is strictly a subset of the second, but acts as a shortcut for some of the common cases.

- **Repeat simplify, reorder, merge**.

- **Back end**. Convert CASEs to their best representation, by
  - splitting into somewhat dense blocks
  - using conditionals where that improves matters
    * a case with a single value to match
    * a case with a single sparse range turns into an if with ORs

  Also, optimize cascased conditionals to ones using AND and OR.

The driver runs all the passes in sequence.

The driver includes a *simulator*, which compares the behavior of the input to the pass with that of the output. It generates test data as follows. For each state variable, it finds the constants against which that variable is compared, to generate that variable's *range set*. Then it adds one more, distinct, value to the range set. Finally, it draws a random element from the range set of each state variable to serve as a test stimulus for the two machines. If any pass produces an output whose behavior differs from that of the input to the pass, the driver throws away the output of the pass, and carries on. This technique proved to be a good way to avoid getting disqualified because of stupid bugs.

The driver also checks for timeout and abandons later passes if it runs out of time. Using the experimental exception-handling mechanism that the Glasgow Haskell Compiler provides (Peyton Jones *et al.* 1999), the driver also catches pattern-match failures and suchlike, abandons the offending pass, and carries on.

On the basis of this entry, the Judges awarded Second Prize to Simon Peyton Jones, Simon Marlow, and Sigbjorn Finne. Their fine results and clear exposition leave no doubts that

Haskell is a fine programming tool
for many applications.

A more complete description may be found at `http://research.microsoft.com/~simonpj/writeup.html`.

First place The first-place winner was the only winning entry that took more than 24 hours to complete. Unsurprisingly, it was larger than the other entries: 3585 lines of Objective Caml. As described by its authors, the entry is a "Frankenstein-style combination of 3 optimizers." Two optimizers work by local rewriting, but the third uses global information. The authors note, and the Judges agree, that "the global synthesizer produces remarkable results." In the final round, Team Camls 'R Us got *first place on all 8 ballots*, with these stunning numbers:

33.1% 89.5% 18.9% 48.8% 87.7% 16.6% 35.2% 36.8%

As stated in the contest announcement, this finish carries "unlimited bragging rights." We have room for only a few excerpts from the extensive report at `http://caml.inria.fr/icfp99-contest`.

Our entry, code-named "Composite," is the outcome of four independent attempts at solving the challenge. Two of those attempts are local optimizers, operating by local size-reducing transformations on the input programs. The other two are global synthesizers that attempt to synthesize from the ground up a program equivalent to the input program, but smaller.

The four optimizers were developed largely independently on Friday, Saturday and Sunday morning. The sub-teams shared basic pieces of code such as parsers and pretty-printers, as well as test suites and equivalence testers. On Sunday afternoon, one of the global synthesizers was very far from working and was abandoned, leaving three working optimizers that each had their own strengths and weaknesses. No optimizer was consistently better than the others on our tests, so we decided to combine them in a single entry, and spent a busy Sunday evening stitching them together.

The first local optimizer is a collection of independent optimizations; each was written quickly once powerful drugs from coffee and chocolate had induced its image in our feverish minds.

- Merging rules with identical statements, exploiting the fact that rules are `CASE` statements for which there is no span cost.
- Merging identical statements in `CASE` and `IF` statements.
- Propagating information from tests in the program, and using it to simplify Boolean expressions and to remove dead code.
- Merging `CASE` statements involving the same variable, possibly combining rules that have two different state lists. The result is evaluated to make sure the cost saved is greater than the possible additional span.
- Replacement of `DECISION n` by `DECISION _` when the current state is known to be `n`.
- Transformation of `CASE` statements into `IF` when appropriate, and removing of degenerate cases (such as those with only a default statement).

This optimizer was completed on Friday evening and submitted as a "lightning entry" under the team name of "Bat8." It placed 6th in the preliminary ranking.

The second local optimizer, developed independently, performs the following transformations:

- Splitting sparse `CASE` statements into cascades of dense `CASE` statements. Because this transformation can duplicate some arms of the `CASE` statement, it must be done with care. Dynamic programming is used to select optimal cut points.
- When beneficial, transforming small `CASE` statements into cascades of `IF`s, and turning large cascades of `IF` tests (over the same variable) into `CASE` statements.
- When beneficial, reordering, which flips the order in which nested `CASE` statements test two variables.
- Factoring of identical arms of `CASE` and `IF` statements. For this purpose, the statements `(DECISION nnn "utterance")` and `(DECISION _ "utterance")` can be converted into one another, using a non-standard form of unification.
- Simplifying Boolean conditions.
- Propagating the information gained from tests already performed into the arms of `IF` and `CASE` statements.
- Combining all the entry points of a description into a single entry point that performs an explicit `CASE` statement on the `state` variable. In itself, this transformation always increases the size of the description, but it may enable a worthwhile reordering transformation later.

The global synthesizer computes a denotational semantics (so to speak) for the input program, then rebuilds an equivalent program by a divide-and-conquer algorithm with backtracking. More precisely, it transforms the program into a list of (Boolean expression, decision) pairs, where the Boolean expression is true if and only if the input program terminates on the given decision. After the Boolean expressions are simplified, the synthesis phase begins.

The program picks an "interesting" variable and a partition of all the values of interest for this variable in the current context. It generates a `CASE` statement on this variable and the lists of pairs (condition, decision) corresponding to each branch of the `CASE`. Each condition is the conjunction of the original condition and the assertion that the variable is in the subset for this case. The conditions are simplified and unsatisfiable clauses are eliminated. The process is repeated recursively until each arm of each `CASE` statement has a single (condition, decision) pair. Now we need to explain how the variable and partition are chosen.

First, we use three heuristics to generate candidate variables and partitions.

- The first heuristic is very simple; try each variable with the maximal partition (i.e. the smallest partition that completely separates the cases, ensuring this variable will not need to be tested again).
- The second heuristic is a bit smarter; for each variable, it tries to find a partition that minimizes the number of duplicated clauses. To this end, it regroups clauses that share many possible values of the variable.

- The first two heuristics can fail to produce a non-trivial partition. The third one acts as a safety net, because it always gives some result. It picks one of the clauses (trying to choose the one with the smallest condition) and generates an `IF` statement (instead of a `CASE` as above), with the clause's `DECISION` in the `THEN` branch, and the program corresponding to the other clauses in the `ELSE` branch.

The controlling heuristic computes an estimated cost for each partition generated by the first two heuristics. The cost function `cut_cost` counts the number of duplicated decisions and divides by the number of branches minus one. The controlling heuristic keeps the 3 best partitions, plus the skeleton generated by the third heuristic, and calls itself recursively.

The two local optimizers are fast and well-behaved; even when iterated, they take only a few seconds on typical inputs. The global synthesizer is exponential by nature, so it's an ideal candidate for eating up whatever remains of the 30 minutes of running time allocated to the programs. We thus run the global synthesizer after the local optimizers, using the smallest of the outputs of the latters as input to the global synthesizer. The output of the global synthesizer can often benefit from peephole optimizations, so we re-run the local optimizers on the output of the synthesizer for a final cleanup.

To help the global synthesizer produce some output in a reasonable amount of time, we start a 10-minute timer at the beginning of the synthesis phase. When this timer expires, the branching factor of global synthesizer's controlling heuristic is reduced from 3 to 1. That is, at each step, the algorithm no longer considers the 3 candidates with the lowest estimated costs, but picks only the best candidate.

Although this excerpt suffices to explain the entry, readers might like to know something about the programming methods used by the winning team.

According to the rules of the contest, the size of the output programs matters a lot. But implicit in the rules is that correctness and timeliness matters even more. We suspected that entries which produced wrong output, or which failed to produce any output within the time limits, would be eliminated. We knew that our program would be buggy, given the short development time and small test suite that we had. Thus, we made every possible effort to ensure that our program would always produce an output semantically equivalent to the input program within the time limits. We achieved this goal by applying the following standard defensive programming techniques:

- At all times, a global variable holds the best result obtained so far. This variable is initialized with a copy of the input program. A timer is set at the very beginning of the execution to one minute less than the time allocated to this run (29 minutes by default). When the timer expires, the program emits the result contained in the global variable. This technique ensures that we always produce an output within the time limits, even if none of the optimizers terminates. This output will be identical to the input program, but that's still much better than no output at all.

- When one of the optimizers produces a result that is smaller than the current best result, the new result is tested for equivalence with the input program before being recorded. The equivalence test proceeds by generating random values for the variables, then evaluating the input program and the new result on this output and checking that they produce the same decision. This process is repeated for 30 seconds. We also had an exhaustive equivalence-testing procedure, but we decided not to use it by fear that it would take too long to complete on large inputs. [A wise decision given the sizes of some state spaces.] For the same reason, we chose to bound the duration of the randomized test rather than its number of iterations. Even though the randomized test is incomplete by nature, we found it extremely effective in catching bad outputs. For example, the global synthesizer produced bad output for two of the test cases used in the final round, but the randomized testing caught and discarded these incorrect results, preventing our entry from being disqualified.

- All exceptions escaping from the local optimizers are caught, causing the program to abort the current optimization round and try the next optimizer in its working list, rather than crashing the whole program. One of us worked on the Ariane 5 code review, and learned the hard way that there are situations where a program must never crash when it encounters an internal error, but rather make all possible attempts to continue working. Whether the program in question runs in a space rocket or on the Judges' machines makes no fundamental difference.

- The only parts of our code that aren't protected by the crash-proofing techniques above are the parser and pretty-printer. We therefore tested them very carefully, making sure that our parser recognizes exactly the input language as defined in the challenge task (including borderline cases such as empty lists of constants or state numbers), and that our printer always produces syntactically correct output.

Now is the time to exercise the "unlimited bragging rights" promised in the contest announcement. Functional languages in general, and the Caml language in particular, were a very good match for the problem. The language features we relied on a lot include:

- Automatic memory management (there was simply no time to waste on explicit memory management);

- Simple and efficient representation of abstract-syntax trees, with support for pattern matching (there was also no time to waste on class-based encodings of abstract-syntax trees or on cascades of elementary tests);
- Strong static typing (caught the usual collection of trivial bugs in each optimizer, and more importantly in their last-minute integration into one program);
- Equally good support for functional and imperative programming (while the general framework of running the optimizers in sequence, record the best result, test for equivalence, handle timeouts, etc., was very imperative in nature, each of the optimizers was written in a purely functional style).

Several features of the Objective Caml implementation also played an important role:

- Fast turnaround, thanks to the bytecode compiler;
- Reasonable set of development tools (we used the parser generator, the replay debugger and the execution profiler—few other implementations of functional languages offer all three);
- Good execution speed, thanks to the native-code compiler (execution speed didn't really matter for the local optimizers, but was significant for the global synthesizer);
- Ability to generate small stand-alone executables that can safely be shipped to the Judges' machines (getting a statically-linked binary was just a matter of adding the `-ccopt -static` to the `ocamlopt` compiler invocation).

The First Prize was well earned by the team of Pascal Cuoq, Damien Doligez, Xavier Leroy, Luc Maranget, and Alan Schmitt. On behalf of the ICFP community, the Judges proclaim that

Objective Caml is the programming tool of choice for discriminating hackers

Readers who are curious about Objective Caml can learn more at `http://caml.inria.fr`.

Conclusions

The ICFP programming contest is an annual event; the 2000 contest will be held in August. We encourage people to enter even if they have no experience with functional programming. We offer special incentives for students, but the ICFP contest is one of the few contests for which *everyone* is eligible. More information can be found at the ICFP web site, at `http://www.cs.luc.edu/icfp`. We hope you will join us this year.

References

Nelson, Graham. 2000. *The Inform Designer's Manual.* Fourth edition. 1652 NW Summit Drive, Bend, OR 97701: Cascade Mountain Publishing. Edited by Gareth Rees. See also the Inform home page at `http://www.gnelson.demon.co.uk/inform`.

Peyton Jones, Simon, Alastair Reid, Tony Hoare, Simon Marlow, and Fergus Henderson. 1999 (May). A semantics for imprecise exceptions. *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 34(5):25–36.

Rees, Gareth. 1995 (August). Christminster. Source code available at `ftp://ftp.gmd.de/if-archive/games/source/inform/minster.tar.gz`. Interested readers are advised to type "Rees Christminster" into the search engine at `google.com`.

Roberts, Michael J. 1996. *The Text Adventure Development System Version 2.0 Author's Manual.* See `ftp://ftp.gmd.de/if-archive/programming/tads/manuals/tadsman.pdf`.