JFP **20** (2): 123–136, 2010.
doi:10.1017/S0956796810000018 First published online 2 March 2010

Experience report: Functional programming in C-Rules

JEREMY WAZNY

Constraint Technologies International, Level 7, 224 Queen Street, Melbourne VIC 3000, Australia
(*e-mail:* jeremy.wazny@constrainttechnologies.com)

Abstract

C-Rules is a business rules management system developed by Constraint Technologies International (www.constrainttechnologies.com) that is designed for use in transport, travel and logistics problems. Individual businesses within these industries often need to solve the same kinds of problems related to scheduling of operations, resource allocation, staff rostering and so on, but each organisation has its own rules and procedures. Furthermore, these problems tend to be combinatorially challenging: before a final solution is chosen, many potential choices need to be generated, evaluated and compared. In C-Rules, users define rules that describe various aspects of a problem. These rules can be invoked from an application, which is typically either an optimising solver or an interactive planning tool. Rules can be used to encode evaluation criteria, such as the legality or cost of a proposed solution, or values like configuration parameters that may be used by the host application to tune or direct its progress. At its core, C-Rules provides a functional expression language that affords users power and flexibility when formulating rules. In this paper we will describe our experiences of using functional programming both at the end-user level and at the implementation level. We highlight some of the benefits of basing our rule system on features such as higher-order functions, referential transparency and static, polymorphic typing. We also outline some of our experiences in using Haskell to build an efficient compiler for the core language.

1 Introduction

C-Rules is a business rules system used primarily to formulate rules for solving scheduling problems in the transport, travel and logistics industries. Consider, for instance, an airline which operates each month according to a (mostly) predetermined flight schedule and staff work roster. Such schedules and rosters must be assembled in accordance with a variety of legal requirements, on top of the airline's own rules and regulations. These rules can be represented within C-Rules and made available to an application, such as a dedicated solver, which can invoke them as necessary. Typically, such an application will generate many potential solutions, using simplified, generic notions of legality (hardcoded into the software), and then make use of more specific rules, defined within C-Rules, to determine the actual validity and cost of those 'solutions'. In general, though, the rule system can return

[1] This article is an expansion of a paper presented at ICFP 2007 in Freiburg.

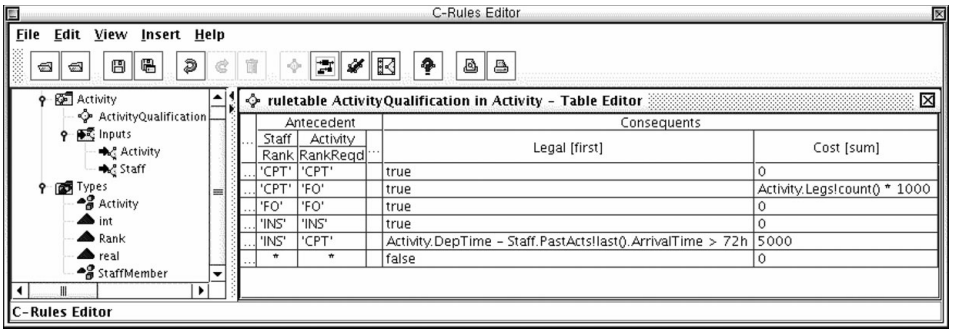

Fig. 1. The C-Rules editor.

values for a variety of purposes, e.g. tuning parameters for an automatic solver or textual advice for a manual operator.

The C-Rules system consists of the following three components, the first two of which we will discuss in the current paper:

1. an editor for users to view, define and edit rules;
2. an evaluation library (Libcrule) for host applications to execute rule queries;
3. a server for storing and tracking rule repositories.

Working within the editor, users define problem-specific rules as tables that produce conclusions (consequents) in terms of some conditions (antecedents.) In this context, a 'rule' or, equivalently, a 'rule table' can be thought of as a problem-specific function.

When producing a table, the user must nominate a set of inputs and outputs and their types. A table consists of a sequence of lines, with each divided into an antecedent and a consequent component. The antecedent acts as a guard: it contains a number of Boolean expressions that, for a given input, determine whether the rule line applies at all. The consequent part of a line contains an expression for every output in the rule table. Rules are executed at the request of a host application, which entails binding input and output sources and evaluating the requested rules. As such, the C-Rules system can be thought of as providing an embedded functional language and associated tools.

Example 1

Figure 1 contains an image of the C-Rules editor. The main panel contains a rule table called `ActivityQualification`, with inputs `Staff` and `Activity` and a pair of outputs `Legal` and `Cost`, which can be seen at the tops of their respective rule table columns.

This table encodes a rule for determining two things: (1) whether a staff member is capable of undertaking a particular activity (the `Legal` output, which is a Boolean) and (2) the 'cost' of allowing this person to perform the activity (the `Cost` output, an integer). This is a simplified version of a rule that an airline might employ when generating a work schedule.

https://doi.org/10.1017/S0956796810000018 Published online by Cambridge University Press

Omitting the details for now, the rule can be understood as follows: The values `CPT`, `FO` and `INS`[2] represent the staff ranks 'captain', 'first officer' and 'instructor', which we have used to distinguish members of an airline's flight crew. The rule described by this table allows any person to undertake, at no cost, an activity which requires somebody of their rank. The second line allows a captain to act as a first officer, with a cost proportional to the number of legs flown. The fifth line of the table permits an instructor to perform as a captain if the activity takes place more than 72 hours after his last activity ended, at a cost of 5,000 units. No other combination of ranks is allowed.

User-declared types can be seen in the left panel and include an enumeration type called `Rank` and record types called `Activity` and `StaffMember`. The type declarations can, of course, be viewed within the editor, but to save space we describe them here. The `Rank` type represents staff ranks and contains the values `CPT`, `FO` and `INS`, described earlier.

An `Activity` record represents a unit of work and contains a number of legs, i.e. flights, (a field named `Legs`), a departure time (`DepTime`), arrival time (`ArrivalTime`) and a required rank (`RankReqd`). Values of the `Staff` type represent staff members and have fields to represent an individual's rank (`Rank`) and a list of previously undertaken activities (`PastActs`). □

A rule table's outputs may be computed independently or in any combination, as required by the host application. Intuitively, evaluation of a single rule table output *o* proceeds as follows:

1. Evaluate each rule line's antecedent part.
2. For every line whose antecedent was reduced to the Boolean true value, evaluate that line's expression for *o*, producing a list of values in the same order as the rule lines appear in the table.
3. 'Coalesce' the results depending on *o*'s coalescing mode. For example, if the coalescing mode is *first*, then only the first result (corresponding to the first matching rule line) is returned. In the *sum* coalescing mode the result is the sum of all of the matching rule lines' values.

Each of a rule table's outputs is evaluated in the same way, though in practice, re-evaluation of common sub-expressions is avoided by our implementation.

Example 2

In the antecedent portion of our example rule table there are two columns, the first of which is headed by `Staff` and `Rank` and the second by `Activity` and `RankReqd`. These represent the `Rank` field of the `Staff` input and the `RankReqd` field of the `Activity` input, respectively. The values in these columns, like `CPT` and `FO`, are shorthand for equality relations on their corresponding inputs. For example, the `CPT` value in the `Staff/Rank` column represents the equality test expression

[2] In our syntax, apostrophes are required around enum literals, which may consist of arbitrary combinations of letters, digits and various punctuation characters. The apostrophes are present in Figure 1, but we omit them from the text.

https://doi.org/10.1017/S0956796810000018 Published online by Cambridge University Press

`Staff.Rank = CPT`, where `Staff.Rank` is the projection of the field `Rank` from the input `Staff`. An `*` in a column represents an always-true value. This is similar to the pattern matching of languages like Standard ML and Haskell, though we allow relations other than equality between inputs and 'pattern values'. In general, additional arbitrary expressions can be added to any line's antecedent.

To evaluate this rule, a value is bound to each of the `Staff` and `Activity` inputs, and the antecedent part of each line is evaluated. Assuming our staff member has a `Rank` of `CPT` and the activity has a `RankReqd` of `FO`, then both the second and last lines of the table will apply. To calculate the `Legality` output, which has coalescing mode *first*, we can simply evaluate the corresponding expression on the second line, in this case the value `true`.

To determine the `Cost` output for the same inputs, we sum the values in the `Cost` column of the second and last lines, since `Cost` has the *sum* coalescing mode. The expression on the second line can be read as 'the number of legs in the input activity, multiplied by 1,000'. This is the cost of allowing a staff member with rank `CPT` to fulfil a role requiring someone of rank `FO`. Note that in this example, only one non-zero cost value could contribute to the sum for any pair of inputs. □

The design of the C-Rules expression language draws on ideas from functional programming, while part of the system itself is written in Haskell, a functional programming language. Our experiences of using functional programming for this sort of problem have been largely positive. The static type system and lack of side effects in the C-Rules language have given users valuable confidence when modifying, checking and debugging rules, and the expressiveness of Haskell allowed us to complete a major new extension to the software in a surprisingly short time.

On the downside, we had to modify the overall design of the Libcrule shared library to accommodate the use of Haskell, and there is the possibility that finding capable Haskell programmers to maintain the existing software may prove difficult. Fortunately, these problems have not had any lasting negative effects, and the productivity benefits have convinced us to employ Haskell for other projects.

In the following sections, we will discuss further our experiences in supporting and using functional programming both at the user level, as a source language within the editor (Section 2), and at the implementation level, in Libcrule (Section 3). We conclude in Section 4.

2 The C-Rules user experience

The C-Rules language and editor are intended to be relatively easy to use for people without programming experience. The editor presents rule tables in a form similar to a spreadsheet,[3] making it more familiar to most prospective users than a text-editor-style interface.

An important goal of the system is to enable rule writers to make minor edits to tables, i.e. without changing the interfaces between tables, with the confidence that

[3] Note that unlike in a spreadsheet, individual 'cells' do not have identities and cannot be referenced. The basic, user-defined 'functional unit' is the rule table. The similarity to spreadsheets is merely visual.

they are not inadvertently introducing errors in other rule tables that may depend on the edited table. We have found that basing the language on a statically typed, pure, functional core, with well-defined interfaces between rule tables and no shared state, has enabled us to provide these sorts of guarantees.

In our experience, catching errors early is especially important in the context of an 'embedded' system because rules will often be invoked many times[4] from host applications that are stateful and whose behaviours may be difficult to follow. The interleaving of host and rule system execution makes bugs in applications extremely tricky to trace. It is also likely that the rules and host application are developed separately, making testing difficult in the first place.

2.1 Expressions

The core expression language, which the user employs when defining the contents of a rule table, is a statically typed purely functional language with call-by-value semantics. The language is essentially the lambda calculus with a large number of built-in operators to make expressions succinct and readable.

C-Rules includes many higher-order polymorphic operators for processing lists and sets of arbitrary values. For instance, built in are `foreach` and `summarise`, which are the well-known *map* and *foldl* from Haskell, and other functional languages.

Many of the operators deal with date and time processing. There are operators for manipulating localised date–time values, i.e. date and time values at some specific location. Associating a location with a date and time is necessary because the length of a day (in seconds) may vary depending on where it is, because of a daylight savings change. Other, more specialised operators, like `maxwindowoverlap`, deal with periods of time and are used by rules to determine things like the maximum number of hours a staff member is working during a week. In general these kinds of operators are particularly useful for transport and scheduling problems, since many rules are concerned with events that occur during a specific period of time.

Many of the provided operators are overloaded, with different implementations based on the types at which they are used. For instance, the `sum` operator can be applied to lists or sets of integer values, floating point values or any other value that can be added. The `+` operator is also defined for meaningful combinations of date, time and duration arguments, e.g. `2007-01-21 18:35 + 12h 30m`, which advances a date–time value by some duration.

Finally, as would be expected, the expression language provides a means to invoke one rule table from another. The system allows users to write explicit rule table thunks which represent a call to a specific table, along with any input values required. These thunks may be placed inside data structures. To apply one, the user must specify which output of the table is to be produced.

[4] A recent profile of Constraint Technologies International's TPAC Rostering application (for airline crew rostering) showed approximately 13.5 million rule table evaluation calls for a small-to-medium-sized problem.

https://doi.org/10.1017/S0956796810000018 Published online by Cambridge University Press

2.2 Types

The C-Rules type system provides most of the basic types one would expect, such as integers, floating-point numbers, strings and Boolean values, as well as date, time and duration types. It also has support for parametric polymorphism and overloading at the expression level but not between rule tables. List and set types are also built in. In addition to these, users can define their own record and enumeration types. Our type system is based on an instantiation of the HM(X) framework (Sulzmann *et al.*1997) and uses finite domain constraints (Marriott & Stuckey 1998) for overloading resolution.

By design, we require that the types of expressions in all cells in a rule table are monomorphic, non-function types and that input and output types are declared by the user. The types of local, lambda-bound variables are inferred automatically and do not require any specific type annotation. Within a single cell, users may make free use of polymorphic, higher-order functions, such as `foreach`, which applies an anonymous function to every element of a list or set.

While limiting the expressiveness of the system, the requirement that types be user declared means that the scope for poor type error reporting is reduced somewhat, as compared with languages in which type declarations are optional, and polymorphic types may be inferred (Wazny 2006). In such a language, a mistake in one definition can be propagated into another before it is reported, making its source unclear. In our more restricted setting, a type error is always localised to the single cell containing the expression that the user last edited.

Though such limitations may seem burdensome to functional programmers used to the expressiveness of a type system like that of Haskell, we have found that rule writers do not feel encumbered by these restrictions. Moreover, of the C-Rules users who are programmers, most who are familiar with statically typed languages, like C or Java, expect to have to declare function (rule table) types anyway. The practice of assigning an input or an output a type is straightforward, since the name and type are presented together in a single graphical dialogue box – there is no separate procedure for assigning a type, which could be forgotten or overlooked.

One distinctive feature of the C-Rule's type system is that it allows for the overloading of record field names and enum values. That is to say any number of user-defined records may have fields of the same name, and similarly, different enum types may contain the same value. It is common for users to create a number of record types, many of which share generic field names, like `Location` or `StartTime`. The user knows which type of record he has in mind when projecting out a specific field, and the system needs to be able to follow.[5] In reality, there is little cause for confusion because monomorphic types must be declared ahead of time for all input variables. Being required to give each field a unique name would be a noticeable hindrance from the user's perspective.

[5] Because rule repositories are stand-alone entities, we can adopt a closed-world approach when disambiguating cases of overloading; if a solution to the typing problem exists, it will be found. The requirement that rule table interfaces are monomorphic reduces the 'search space' considerably in practice.

https://doi.org/10.1017/S0956796810000018 Published online by Cambridge University Press

2.3 Other features

The C-Rules editor can also statically detect other kinds of problems in, and between, rule tables and their expressions. As well as editing the contents of rule tables, users may modify other aspects of the repository, such as inputs and outputs or the definitions of user-specified types. These sorts of changes to table interfaces or global definitions may introduce errors. For example, removing a field from a record type definition will invalidate all occurrences where such a record is constructed with this (now-invalid) field. The editor is able to detect and present these sorts of errors to the rule writer and even suggests likely fixes. These suggestions can be applied at the press of a button. We have found automatic and interactive problem resolution to be a necessity, since individual rule writers may not be familiar with an entire repository.

A simple interpreter built into the editor allows users to define and run tests. Because rules have no state and perform no side effects, running them outside of the context of their usual application is never a problem.

2.4 Use of C-Rules

Feedback has shown that users within the organisation as well as external customers have reacted positively to the C-Rules editor and have adopted a functional style of rule writing without any problem. C-Rules is used by customers to describe pairing (TPAC Pairing 2007) and rostering (TPAC Rostering 2007) problems for optimising airline schedules.[6] An average-sized rule repository for an airline client consists of about 220 rule tables and a total of 1,200 rule lines between those tables. This represents a significant body of knowledge of a company's business operations, the correct encoding and application of which is vital to the successful use of the software.

In most cases C-Rules is delivered alongside another application, as mentioned above, and provides a means for the client to configure its results. Besides that, the company has also had customers who have deployed C-Rules to validate manually created solutions to their own transportation problems. Within Constraint Technologies International, C-Rules has also been employed to manage less exotic tasks. For instance, we make use of a logging library – for recording the progress of an application as it runs – which is configured using C-Rules. The end user can write rules which control whether or not information is logged on the basis of the status of the application, as passed to the rules.

For complex problem domains, the benefits outlined in this section are especially welcome. Although we do not explicitly use the term 'functional programming'when promoting the software, nor do we make a point of the implementation language, the value of the features we have described are appreciated by customers.

[6] In aircraft scheduling, *pairing* is the problem of assembling individual flights into cost-effective, sequences to be flown by a single crew. *Rostering* is the process of assigning actual staff to these flights.

https://doi.org/10.1017/S0956796810000018 Published online by Cambridge University Press

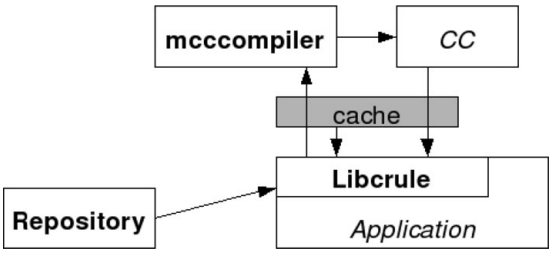

Fig. 2. Libcrule system diagram.

3 Implementation of the C-Rules compiler

To evaluate rule tables, applications use the Libcrule library, which is capable of loading, compiling and executing rules. The current implementation of the library itself is written in C++ but makes use of a separate executable for compiling rule repositories. This compiler, called *mcccompiler*, is written in Haskell and produces ordinary ANSI C code from the internal rule repository representation and an input–output binding description.

An application written against the Libcrule API will perform the following steps to evaluate a rule table:

1. Create a new library instance, based on a single rule repository.
   - The repository may be sourced from a file or from a repository server.
2. Start a new session.
   - A session represents a repository and a set of input–output bindings.
   - Multiple sessions, based on the same repository, may be run concurrently.
3. Create input and output bindings.
   - This amounts to describing the location and representation of values to be read and written in the client application's address space.
4. Check and optimise.
   - The parts of the repository bound to by the application are statically checked and optimised by routines in the Libcrule library.
5. Compile to executable form.
   - The optimised representation is transformed to C code by mcccompiler.
   - The object code is then generated by a C compiler.
6. Write values to the bound input locations, and request evaluation.
   - This can be repeated as many times as necessary without performing any of the previous steps.

Figure 2 contains a diagram illustrating the relationship between Libcrule, the host application, the rule repository and compilers. In brief, an application makes use of Libcrule to load a rule repository, which it then processes and feeds to mcccompiler. The result of that compilation step is passed to a C compiler CC, which generates an object file that is dynamically loaded into Libcrule. A cache sitting between Libcrule and the compilers serves to prevent unnecessary recompilation.

https://doi.org/10.1017/S0956796810000018 Published online by Cambridge University Press

3.1 History

An earlier version of Libcrule used a backend which produced and interpreted a purpose-built byte code language. The byte code compiler and interpreter were written by a single developer, on and off, over a period of about a year. Both were implemented in C++ and occupied approximately 25,000 lines of code. The compiler and interpreter were both integrated directly into the Libcrule shared object – no external executable was invoked.

When we found that the cost of interpretation was unacceptably high for our applications, we decided to adopt the approach of compiling to machine code via C and then linking and directly executing the result.

We chose C as a target language for a number of reasons. Firstly, C allows us to produce fairly low-level code with predictable performance characteristics. It is usually straightforward to determine, or to have some intuition about, whether a fragment of C code can be executed efficiently. As is the common practice, we treat C as a kind of portable, high-level assembler.

Further, C affords us great control over the representation of data in memory. This is important because our applications tend to manipulate fairly large data sets. Using techniques such as embedding Boolean and enum values as bit fields can save us a lot of memory. A recent review of one of the largest rule repositories showed user-defined record types with up to 14 Boolean fields and multiple enum fields, all of which can be encoded as a single 32-bit word. When hundreds of thousands of such records are created, the savings can become significant.

Finally, C compilers are available on all of the systems that our software targets. This means that we do not need to bundle a C compiler with C-Rules, which would be a burden. We did briefly consider targeting higher-level languages such as Haskell and Mercury (Mercury 2007) but excluded them for this and other reasons.

In 2006, the author started work on a design and, in parallel, began implementing a prototype in Haskell, to test the design's viability. Before this step, we had not given much thought to the implementation language and had assumed the compiler would ultimately be implemented in C++, as was the rest of the library. By the time the design was done, however, the prototype was generating useful code for a number of small test cases; so we decided to simply refactor and extend it into a full implementation. It took approximately a month to complete the compiler implementation and to integrate it into Libcrule. At that stage it was passing all of our existing regression tests and was providing equivalent functionality to the system it had replaced. The author spent the following month identifying and fixing performance problems in the generated code – exceeding the sophistication of the original backend.

We have found the experience of iterative design and prototyping to have been extremely beneficial. The drive to develop a concise and accurate design specification coupled with the additional foresight gained by having a working implementation greatly reinforced each other. The prototype implementation was able to inform the design, while the design kept the implementation focused and manageable.

3.2 Implementation

Having a strong, static type system in the source language has enabled us to use efficient data representations in the generated code. Because the types of expressions are known at compile time, run-time values can remain untagged, and multiple record fields can often be packed at the bit level, when necessary. The Libcrule library is capable of specialising the implementation of overloaded operators depending on the user-declared types in use. The lack of side effects similarly makes feasible a large host of optimisations, such as common sub-expression elimination, partial evaluation (Augustsson 1997), data structure fusion and other techniques specialised for our particular rule structure.

The table structure of rules allows us, in certain circumstances, to find sets of lines which are disjoint, i.e. can never apply at the same time because their antecedents cannot both evaluate to *true*. We can take advantage of this to speed up the process of determining which rule lines can actually contribute to an output. When possible, mcccompiler will generate C `switch` statements and supporting data structures, in preference to code which checks each line of a table in turn.

Most of these optimisations are performed by Libcrule as source-to-source translations of its internal rule repository representation. These occur during Step 4 of the process presented in Section 3 and are independent of the code generation stage. Indeed, the previous C++ backend implementation made use of the same optimised internal representation. The task of the code generator, mcccompiler, at Step 5, is to transform this into a form that is suitable for execution. In our case, that means C program text, which can then be further compiled into binaries.

One of the factors which made programming in Haskell especially productive was its ability to easily support embedded domain-specific languages. Our implementation makes use of two such languages: one for representing the compiler's output and another for defining compilation schemes for operator expressions. Although there is much more to the implementation, we will focus our discussion on these aspects, since we feel they were the areas in which we gained most from the use of Haskell.

The compiler's intermediate target language is the untyped lambda calculus with C language specific layout combinators, inspired by the pretty printing library of Hughes (1995). Evaluating, i.e. executing, a program in this intermediate language yields plain text C code. We chose this unusual target because, initially, we had simply assumed that the compiler's implementation language would be C++ – the language used for the rest of Libcrule – and wanted the ability to generate parameterised, and therefore reusable, fragments of the target language. If we had planned to use Haskell from the beginning, we could have simply made use of its own facilities for abstraction, rather than embedding lambdas into our target language and then having to interpret them.

Figure 3 contains a small sample of the available layout combinators. The `<>` and `$$` binary operators allow two fragments of code to be placed beside and above each other, respectively. The `$>` operator places one fragment above an indented version of the other. Many other combinators, some directly corresponding to the syntactic

https://doi.org/10.1017/S0956796810000018 Published online by Cambridge University Press

```
a <> b = ... -- a beside b
a $$ b = ... -- a above b
a $> b = ... -- a above an indented b
block e = "{" $> e $$ "}"
paren e = "(" <> e <> ")"
while c b = "while" <> paren c $$ block b
```

Fig. 3. A sample of layout combinators for producing formatted output.

elements of the target language, can then be straightforwardly defined in terms of these building blocks.

By using these combinators, the description of generated code remains concise and maintainable, while the actual text generated is uniformly readable. The expression, `while "A" "B"` is rendered as

```
while(A)
{
  B
}
```

We also designed and used a simple rule-based rewriting language for transforming operator expressions into their intermediate language implementations. Having this sort of representation made it easy to specify compilation schemes for the large number of operators provided by the user-level language. It also allowed us to mechanically generate rules for compiling many operators from concise specifications.

Each rule has three parts: a head which consists of a typed operator expression, a guard which contains predicates and a body which represents the code to generate in the target language. The compiler contains a list of these rules, which it consults whenever compiling an operator expression. In short, a rule can be used to compile an expression when that rule's head matches the expression, and all predicates are satisfied. Part of the compiler's implementation is an interpreter for this rule language.

The following is a simple example of a compilation rule that the system employs. It consists of a rule head and a conditional guard (to the left of the $\rightsquigarrow$) and a compilation result (to the right):

$$not : bool \rightarrow bool \mid true \rightsquigarrow (\lambda x.!x)$$

This rule describes the compilation of the Boolean not operator in the expression language into our intermediate lambda calculus-with-C combinators language. In brief, this rule says that the operator *not*, with type $bool \rightarrow bool$, can be compiled into a function, in our intermediate language, which places an `!` in front of its argument. The symbol `!` is simply the Boolean not operator in C. In this rule, the guard is simply *true*.

Below is a fragment of Haskell code which encodes this rule in our implementation. We show it simply for illustrative purposes and omit the details of the various helper-functions/variables it employs:

```
crule (opNot, op[bool, bool]) [] (lam x ("!" `app` x))
```

https://doi.org/10.1017/S0956796810000018 Published online by Cambridge University Press

The `crule` function constructs a rule consisting of a typed head `(opNot, op[bool, bool])`, an empty predicate list `[]` which corresponds to the *true* and a body `lam x ("!" ‘app‘ x)` which reflects the lambda expression of the original. The `app` and `lam` functions correspond to function application and lambda abstraction expressions, while `x` names a value which the rule interpreter understands to be a variable.

The rule language has built into it a variety of predicates which can be placed in the guard to allow for more expressive rules. They include, amongst others, the typing relation $x : t$, which is true when expression x has type t and the compiles-to relation $\rightsquigarrow$ itself.

The following, slightly more complicated rule describes compilation of the not-equals operator applied to two arguments, x and y:

$$neq(x, y) : bool \mid (eq(x, y) : bool \rightsquigarrow E_1,\ not : bool \rightarrow bool \rightsquigarrow E_2) \rightsquigarrow E_2 E_1$$

The rule states that the result of compiling $neq(x, y)$ is the expression $E_2 E_1$ (E_2 applied to E_1) in our intermediate language, where E_1 is the successful result of compiling the expression $eq(x, y)$ (the equality operator applied to x and y) and E_2 is the implementation of *not*, as described by the earlier rule.

An encoding of this rule in our implementation follows:

```
crule (opNotEqual v1 v2, bool)
      [(opEqual v1 v2 ‘as‘ bool) ‘compilesTo‘ c1,
       (opNot ‘as‘ op[bool, bool]) ‘compilesTo‘ c2]
       (c2 ‘app‘ c1)
```

In the Haskell version of this rule, the `as` and `compilesTo` functions stand in for the typing relation and compiles-to relation $\rightsquigarrow$, respectively, while `app` is function application (implicit in the rule description). The identifiers `v1`, `v2`, `c1` and `c2` are values that the compilation-rule interpreter will recognise as variables.

Using these and other predicates, we are able to describe the compilation of most of the C-Rules language's operators with such rules.

3.3 Analysis

Overall, the benefits of using Haskell to develop our compiler, as compared with our previous C++ solution, were significant.

- Less code: The compiler is implemented in about 9,300 lines of code while the original backend is approximately 25,000 lines of C++ (omitting comments and blank lines.) Note that the C++ backend contains a separate byte-code compiler and interpreter, while the Haskell backend contains only a compiler. We estimate, though, that an interpreter and compiler written in Haskell, in the style of the C++ backend, would not require much more code than the existing Haskell backend; probably on the order of a few thousand lines.
  A large part of both implementations consists of implementations of the many built-in operators available in the source language. The core of our compiler closely matches its specification.

https://doi.org/10.1017/S0956796810000018 Published online by Cambridge University Press

- Fast development: Most of the design and implementation was completed by one developer over the course of a single month. The system was subsequently extended, beyond the capabilities of the C++ backend, and its results were optimised for another month before being deployed. By comparison, the C++ backend took an individual developer almost a year to complete. Because of the straightforward semantics of our source language, the C++ implementation did not inform the design of the new compiler in any significant way.
- Easier testing: Being able to run the compiler within a Haskell interpreter allowed us to easily test and evaluate it during development. If we had chosen C++, testing would have required a great deal more effort, meaning that in practice, much of the code would have remained untested until the end of the programming phase.

In general, the language features in Haskell provided a better match for the problem we were trying to solve than those of C++. For instance, the core expression language can be concisely represented in 17 lines of algebraic data type definitions in Haskell. On the C++ side, 10 classes, in 10 separate files, are used to represent the same language. Granted other designs are possible, but having separate classes for different types of expression seems to be the natural solution in an object-oriented language. An unfortunate consequence of this is that the implementations of functions which apply to expressions in general have their implementations split into a collection of separate functions – one per C++ class – making them difficult to read and maintain.

Despite all of the advantages outlined above, we have encountered a number of problems in our use of Haskell. Though they may seem relatively minor, in reality any one of them could be enough to dissuade an organisation from using Haskell in their products.

- No Haskell implementation that we are aware of at the time is capable of generating relocatable code. This means that our compiler must be a separate executable. This is unfortunate because, previously, the entire library could be shipped as a single shared object file.
  Because of this, we settled on a design in which the compiler's input is a serialised version of the source program, which it must then parse, adding unnecessary overhead.[7]
- The number of skilled Haskell programmers available, i.e. those with a decent amount of practical experience working on non-trivial programs, is tiny. Efforts to mitigate this risk through cross-training have been made, but in a commercial environment, a persistent skills shortage could ultimately necessitate a (reluctant) rewrite in another language.
- Because of the speed of its generated code, Glasgow Haskell Compiler (GHC 2007) is the compiler of choice when developing applications in Haskell, but

[7] One upside of this is that if anything goes wrong during code generation, we have a perfect snapshot of the program just before the failure, which can be easily replayed.

it can be difficult to install properly on a platform for which no binary is available.

Overall, we were pleased with the level of productivity that we were able to achieve using Haskell and have used it subsequently in other, related projects.

4 Conclusion

Functional programming languages and techniques have proven invaluable in the design and development of C-Rules, at both the user level and the implementation level.

Static typing, referential transparency and decoupling of code allow the system to provide strong guarantees to users about the safety of their rule definitions, while also encouraging testing and facilitating debugging. These benefits have allowed both developers within the company and customers to specify their problems in clear, well-understood terms.

Deciding to implement a compiler for our core language in Haskell has also paid off greatly. The final design and implementation match each other closely, aiding maintenance and testing, and the fact that we were able to meet our requirements within about a month gave us time to spend optimising the compiler's results.

Acknowledgments

Thanks to Karl Anderson, Michael Sperber and the anonymous reviewers for their valuable comments and suggestions.

References

Augustsson, L. (1997) Partial evaluation in aircraft crew planning. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*. New York: ACM, pp. 127–136.

GHC. (2007) Glasgow Haskell Compiler [online]. Available at: `http://www.haskell.org/ghc/` Accessed July 2007.

Hughes, J. (1995) The Design of a pretty-printing library. In *Advanced Functional Programming*, Jeuring, J. & Meijer, E. (eds), LNCS vol. 925, Heidelberg, Germany: Springer, pp. 53–96.

Marriott, K. & Stuckey, P. J. (1998) *Programming with Constraints: An Introduction*. MIT Press. Cambridge, Mass.

Mercury. (2007) Melbourne Mercury Compiler [online]. Available at: `http://www.cs.mu.oz.au/research/mercury/` Accessed July 2007.

Sulzmann, M., Odersky, M. & Wehr, M. (1997) Type inference with constrained types. *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL 4)*, Paris, France.

TPAC Pairing. (2007) TPAC Pairing Optimizer [online] . Available at: `http://www.constrainttechnologies.com/components/XBO_Pairing_Optimiser.html` Accessed July 2007.

TPAC Rostering. (2007) TPAC Rostering Optimizer [online]. Available at: `http://www.constrainttechnologies.com/components/CAM_sharp_Rostering_Optimiser.html` Accessed July 2007.

Wazny, J. (2006) *Type Inference and Type Error Diagnosis for Hindley Milner with Extensions*. PhD thesis, University of Melbourne, Victoria, Australia.