Partial evaluation in aircraft crew planning

Lennart Augustsson

Carlstedt Research & Technology
Stora badhusgatan 18-20
S-411 21 Göteborg, Sweden
Email: augustss@carlstedt.se
WWW: http://www.carlstedt.se/~augustss
Phone: +46 31 701 42 36
Fax: +46 31 10 19 87
and
Department of Computing Sciences
Chalmers University of Technology
S-412 96 Göteborg, Sweden
Email: augustss@cs.chalmers.se
WWW: http://www.cs.chalmers.se/~augustss
Phone: + 46 31 772 10 42
Fax: + 46 31 16 56 55

Abstract

In this paper we investigate how partial evaluation and program transformations can be used on a real problem, namely that of speeding up airline crew scheduling.

Scheduling of crew is subject to many rules and restrictions. These restrictions are expressed in a rule language. However, in a given planning situation much is known to be fixed, so the rule set can be partially evaluated wit respect to this known input.

The approach is somewhat novel in that it uses truly static input data as well as static input data where the values are known only to belong to a set of values.

The results of the partial evaluation is quite satisfactory: both compilation and running times have decreased by using it. The partial evaluator is now part of the crew scheduling system that Carmen Systems AB markets and which is in use at most of the major European airlines and in daily production.

Keywords: Partial evaluation, program transformation, generalized constant propagation, airline crew scheduling.

1 Introduction

Next to fuel costs, crew costs are the largest direct operating cost of airlines. In 1991 American Airlines reported spending $1.3 billion on crew [AGPT91]. Other major airlines have similar costs. Therefore much work has been devoted to the planning and scheduling of crews over the last thirty years.

The planning of aircrafts and crews in large airlines is a very complex problem, see [AHKW97] for a good overview. To make the problem tractable it is normally divided into four parts.

Construct timetable
: First, the time table is produced with the objective to match the expectations of the marketing department with the available fleets and other constraints. The output of this process is a number of legs (non-stop flights) which the airline decides to operate.

Fleet assignment
: Second, aircrafts are allocated to the legs. Again there must be a match between expected number of passengers and goods and the available aircraft fleets. The output of this problem is the timetable augmented with aircraft information.

Crew pairing
: Third, pairings are constructed. A pairing is a sequence of flight legs for an unspecified crew member starting and ending at the same crew base. Such a sequence is often called a CRew Rotation (CRR). The crew member will normally be working on these legs, but a pairing may also contain legs where the crew member is just transported. Such a leg is called a deadhead. Legs are naturally grouped into duty periods (working days) called RoTation Days (RTD). Each rotation day is separated by a lay-over (an overnight stop). Legal pairings must satisfy a large number of governmental regulations and collective agreements which vary from airline to airline. The output of this phase is a set of pairings covering the legs in the timetable.

Crew assignment
: The fourth planning problem is to assign pairings to named individuals. This is the crew assignment or rostering problem. The objective is to cover the pairings from the previous stage as well as training requirements, vacations, etc. while satisfying work rules and regulations.

Carmen Systems AB markets a system that handles the last two parts of the planning problem. The system is in use in most of the major European airlines. The optimiser runs in stages, first it generates a large number of possible pairings (using some clever heuristics [TE96], second it selects a subset of these that covers all the legs that should be scheduled, and does it to a minimal cost. The

second part is a huge set covering problem, [Wed95]. Each covering (potential solution) needs to be checkedso that if fulfils all the rules mentioned above. A run of the optimizer is a computationally heavy job; a run may take from a few minutes to a few days of computer time. It is important to make this a fast as possible, since this enables more tries to find a good solution to be made (finding good solution usually requires some manual "tweaking" and rerunning the optimizer). Testing of rule validity usually takes up the largest part of the computation time so any improvement of it will be beneficial.

The rules in the Carmen system are expressed in a special language (described below). A complete rule set is usually rather large, containing thousands of lines, because it covers all fleets, long haul, short haul, all crew categories, different union agreements, scheduling standards, etc. However, when a particular problem is to be solved, only a subset of the rules are really needed, because the problem may only be for a particular crew category, on a specific aircraft type etc.

So we have a large program (the rule set) where some of the input data is known (such as the crew category etc), this spells: *partial evaluation.*

2 The rule language

To express the rules that affect the legality and costs of different solutions to the planning problem, Carmen Systems has developed a proprietary language, the Carmen Rule Language (CRL), [Boh90]. CRL can be said to be a pure, "zeroth order", strongly types, functional language. "Zeroth order" because the language does not have any functions, except a few built in ones. The evaluation of the rules also uses a form of lazy evaluation, but that is of no consequence for us.

A leg has a number of attributes. Some of these attributes, e.g., *aircraft type*, *arrival time*, and *departure time*, are given for each leg. Other attributes, e.g., *length of flight*, are computed from the given ones. It is these calculations that the rule language describes.

The input to a rule evaluation is a set of pairings, i.e., a number CRRs where each CRR contains of a number of RTDs. Each RTD contains a number of legs. The task of a rule evaluation is to find out if the given set of CRRs is legal or not. (If it is legal, a cost is also computed, but that is less important here.) In each of the legs, RTDs, and CRRs some attributes are given, but most of them are computed from other attributes.

In our examples we will use a different syntax than the real language uses to (hopefully) make things clearer. To define externally given attributes we will write "`level.attribute :: type;`" where the level is leg, RTD, or CRR. Attributes defined in the language will be written "`level.attribute :: type = expression;`". Both the level and the type of the definitions are not really necessary since they can be deduced, but they are given for clarity. Such a

program	::=	{*definition*}
definition	::=	*constant-definition*
		level-definition
		rule-definition
constant-definition	::=	*name* :: *type* = *expression* ;
level-definition	::=	*level* . *name* :: *type* = *expression* ;
rule-definition	::=	`rule` *level* . *name* = *expression* ;
type	::=	`Int` | `Bool` | `String` | `AbsTime` | `RelTime`
level	::=	`leg` | `RTD` | `CRR`
expression	::=	*literal*
		name
		expression binop expression
		`if` *expression* `then` *expression* `else` *expression*
		aggregate(*level* , *expression*) [`where`(*expression*)]
		built-in-function(*expressions*)
		`case` *expressions* `of` {*case-block* `otherwise`} *case-block* `endcase`
		`case` *expressions* `external` *string-literal*`endcase`
case-block	::=	{*pattern* , } *pattern* `->` *expression* ;
pattern	::=	*literal*
		literal `..` *literal*
binop	::=	+ | − | * | / | `div` | `and` | `or`
expressions	::=	{*expression* , } *expression*
aggregate	::=	`sum` | `any` | `all` | `count`
		`next` | `prev` | `first` | `last`

Figure 1: Simplified grammar for CRL.

definition means that for each item on the given level (leg, RTD, or CRR) an attributed will be computed according to the formula.

A rule set, or program, defines a number of attributes and rules. The rules are boolean expressions preceeded by the keyword `rule`. Checking if a solution (i.e., a grouping of legs into RTDs and CRRs) is legal amounts to checking that all the rules evaluate to true.

The types and operators of the language are fairly limited and mostly self-explanatory, so we will give no formal description of them except for the simplified grammar in figure 1. Two unfamiliar types are probably `AbsTime` and `RelTime`. They represent absolute times and time differences respectively.

The aggregate functions evaluate an expression a lower level and aggregate them. A where clause acts as a guard and makes it possible to only aggregate those expressions on the lower level where the guard is true. E.g., "`sum(leg,`

```
leg.departure :: AbsTime;     -- The time the plane departs.
leg.arrival :: AbsTime;       -- The time the plane arrives.
leg.aircraft_type :: String;  -- Type of plane, e.g. "747"
leg.deadhead :: Bool;         -- Deadhead leg

leg.blocktime :: RelTime = arrival - departure;

rule leg.short_legs =
    blocktime < 4:00;         -- No more than 4 hours per flight
```

Figure 2: A simple example.

`x) where(y)`" will compute (for an RTD) the sum of all the `x` attributes of the legs where the `y` attributes are true.

The language has been carefully designed so that it is impossible to write non-terminating computations and there is no way a program can fail.[1] These properties make the rule language very amenable to program transformations since $\perp$ cannot occur.

2.1 Rule example

Figure 2 shows a simple rule set that expects the departure, arrival, and aircraft type to be supplied externally for each leg, whereas the blocktime[2] is computed for each leg. There is also a rule which states that the blocktime should be less than 4 hours for each leg. There is no need to quantify the rule expression because is it automatically tested wherever it applies.

In the extended example, (figure 3) other leg attributes are defined and `total_worktime` is defined on the RTD level. It is defined as the sum of `worktime` for each leg belonging to the RTD. `Sum` is one of a few built in functions that aggregate data on a lower level in the hierarchy. Remember that the hierarchy is wired into the problem, and thus into the language. Therefore there are no ambiguities as to which legs to sum over for any given RTD. Note how the second rule refers to both an attribute at the leg level and at the RTD level. Since each leg belongs to exactly one RTD, there is again no ambiguity as to what RTD to use in the computation of the rule.

[1] A program can in principle divide by 0, but we were told to ignore this. Divisions are very rare in "real life."

[2] Blocktime is the time from when the wheel blocks are removed at departure until they are replaced at arrival; it is considered the total time of the flight.

```
leg.briefing :: RelTime =
    if aircraft_type = "747" then 1:00 else 0:30;
leg.debriefing :: RelTime = 0:15;

leg.worktime :: RelTime = briefing + blocktime + debriefing;

RTD.total_worktime :: RelTime = sum(leg, worktime);

max_total_worktime :: RelTime = 9:00;

rule RTD.short_days =
    total_worktime < max_total_worktime;

rule leg.even_legs =        -- No leg may be longer than half
    worktime < total_worktime / 2;     -- the work in a day.

RTD.deadheads_per_rtd :: Int = count(leg) where(deadhead);

CRR.deadheads_per_crr :: Int = sum(RTD, deadheads_per_rtd);

rule CRR.few_deadheads =    -- At most 2 deadheads in a pairing
    deadheads_per_crr < 3;
```

Figure 3: Extension of the simple example.

```
leg.briefing :: RelTime =
    case time_of_day(departure) of
      00:00 .. 06:00  ->  0:15;
      06:01 .. 18:00  ->  0:30;
      18:01 .. 23:59  ->  0:20;
    endcase;

leg.minimum_ground_stop :: RelTime =
    case arrival_airport_name, deadhead, day_of_week(arrival) of
      "FRA", False, 1 .. 5  ->  1:00;
    otherwise
      _,     False, 1 .. 5  ->  0:45;
    otherwise
      _,     _,     _       ->  0:30;
    endcase;
```

Figure 4: Some table/case expressions.

2.2 Tables

A lot of rule computations can be simply expressed as table lookups. CRL has a fairly powerful case construct to express this. In the case construct several values (keys) are matched simultaneously against a set of patterns. Each pattern can be a literal or a range. The patterns are only allowed to overlap if marked so explicitly. Figure 4 shows some simple examples of these table constructs. The first one determines a briefing time depending on the time of the day of the arrival (arrival is a AbsTime, but the `time_of_day` function drops the date part of it). The second table states that the minimum ground stop is one hour if the arrival is in Frankfurt (FRA) on a non-deadhead leg on a Monday thru Friday, etc.

2.2.1 External tables

To avoid repeated recompilations of a rule set, and to make it more flexible, there is a concept of external tables. An external table is like the case construct just described, but the case arms are not given in the rule file. Instead they are loaded dynamically from a file at run time. Because the case arms are not available to the compiler, this construct is less efficient.

2.3 Void

Although CRL is defined in such a way that everything terminates and there are no failures, there is a small complication. To handle some functions (and sometimes given attributes) that can fail there is a concept of exception, called *void*. A typical expression that could give rise to *void* is "`next(leg, x)`" which is supposed to give the value of the `x` attribute of the next leg (relative to the leg where it is computed) within an RTD. But for the final leg in an RTD it does not exist, so it gives the "value" *void*. If a *void* occurs in an expression it will be propageted to the result. *Void* propagates until caught with a special language construct.

3 Program transformations

The transformations done to the program really fall into three categories. First, the transformations that can be performed given just the program. These transformations are like constant propagation and constant folding. Second, there are the transformations that can be made given the static data from the problem. The static are most what are called *parameters*. A parameter is like a global rule constant, but it is not given at compile time, but instead at run time. External tables are also considered to be static data. The third category is those transformations that can be performed because all the legs are known.

3.1 Constant propagation and folding

A typical rule set contains many definitions that are merely constants. These constants can, of course, be substituted into the code and constant folding can be applied. This step can do much more work than one might imagine at first.

The constant folding does not only handle built in functions, but it also includes matching in case expressions. If one of the keys turns out to be a constant that means that a column of the case expression can be removed together with all the rows that did not match the key.

3.1.1 External tables

External tables are turned into internal tables by the partial evaluator, because internal tables are more efficient. The semantics of external tables is unfortunately somewhat different from the internal tables. In an internal table no overlap is allowed between entries unless explicitly stated, however for external tables the entries are matched from top to bottom. This means that when an external table is converted to an internal one it cannot be used as is. The overlap annotation could be inserted between each entry in the resulting table, but in doing so much efficiency would be lost since the rule compiler handles non-overlapping entries much more efficiently.

```
leg.debriefing :: RelTime = expr;
```

Before deadhead cloning.

```
leg.debriefing :: RelTime =
    if deadhead then debriefing__dh else debriefing__ndh;
leg.debriefing__dh  :: RelTime = expr;
leg.debriefing__ndh :: RelTime = expr;
```

After deadhead cloning.

Figure 5: A simple example.

Therefore the external table needs to be transformed to remove overlaps when it is converted to an internal table. This processing is complicated by the fact that in each column of a table there can be a range as well as a literal. In fact if we have a table with n columns each table entry can be thought of as defining an n-dimensional box, i.e., a rectangular parallelopiped, that defines what it matches in the n-dimensional space defined by the columns. This is the key insight in the transformation of the external tables.

As each new table entry is processed (top-down) its box is computed and from that all the boxes from the preceding entries are subtracted. The result of subtracting one box from another can always be expressed as a sum of boxes. After subtracting all preceding boxes we are left with a (possibly empty) set of boxes that correspond to non-overlapping entries with the same right-hand-side. These boxes are converted to entries which are added to the internal table.

3.2 Cloning

One of the features of partial evaluation is that it generates a different specialisation of the same function for each distinct argument. In CRL there are no functions, so the same kind of function specialisation does not carry over directly. The CRL partial evaluator also does some specialisation, but it is very ad hoc.

Deadhead flights are not really part of the normal flight you want to plan, since they are only used as transport. This means that many attributes are computed in a very different way for deadhead legs compared to ordinary legs. Furthermore, the leg set contains much more precise information if the deadheads are not considered. The reason being that even if the problem at hand is to plan for a particular aircraft fleet, deadheads may happen on completely different fleets, or even on different airlines.

To handle deadheads in a good way they are treated specially during the transformation. Each definition on the leg level is split into two part, one where

it assumed that `deadhead` is true and one where it is assumed that `deadhead` is false. See figure 5 for an example. The definition of the original attribute (`debriefing`) is inlined everywhere it occurs. The two new definitions will optimized under the assumtion that `deadhead` is true (resp. false). The inlined `if` expressions will very often reduce to one of the branches because the value of `deadhead` is known in the local context.

3.3 Set based evaluation

In any given planning situation there is a set of legs that should be arranged into RTDs and CRRs. This means that we actually have a lot of information about the values of the externally supplied leg attributes. These leg attributes can only take on the values of the leg attributes as they appear in the flight plan.

This is really valuable information, considering many of the attributes only take on one or two different values. The reason for this is that a flight plan normally covers only a week or two of time, a limited geographical area, and a few aircraft types.

The practical consequence of this that an expression like "`if aircraft_type = "747" then x else y`" can be simplified to "`y`" if the flight plan contains no 747 aircrafts.

These transformations are quite frequent because a rule set typically covers all of the airline's operation, so it has information about sort haul and long haul, cockpit and cabin personnel, all times of the year, etc. But a given flight plan will only use a subset of the rules because it is only about e.g., short haul cabin personnel during one week in August.

The set based transformations are based on a set evaluator. The set evaluator is like an ordinary evaluator for expressions in the language, except that the environment contains sets of values for the variables instead of single values, and that the result of evaluation is again a set. A fragment of the set evaluator is shown figure 6. The sets computed by the set evaluator are always as large or larger than the real set of values an expression can have.

The only interesting part of the set evaluator is the handling of `if`. The most naïve way would be to just take the union of the sets from the two branches. Slightly more sophisticated is to check if the condition is a singleton true or false and pick the corresponding branch if it is a singleton and otherwise take the union. However, in this case, there is actually additional information within each of the branches; the condition is known to hold or not to hold.

This means that when evaluating "`if x=5 then e1 else e2`" within the `then` branch, `x` is known to be 5, and within the else branch it is known not to be 5. The use of these facts account for the change of environment in the evaluation of the branches. The environment modification function looks for relational operations on variables to modify the environment. You can imagine a much more much sophisticated analysis which keeps track of the condition

$$
\begin{array}{lcll}
\mathcal{S}[\![\, x \,]\!]\rho & = & \rho x & \\
\mathcal{S}[\![\, l \,]\!]\rho & = & \{\mathcal{L}[\![\, l \,]\!]\} & \text{a literal} \\
\mathcal{S}[\![\, e_1 \texttt{+} e_2 \,]\!]\rho & = & \{x + y \mid x \in \mathcal{S}[\![\, e_1 \,]\!], y \in \mathcal{S}[\![\, e_2 \,]\!]\} & \\
\mathcal{S}[\![\, e_1 \texttt{=} e_2 \,]\!]\rho & = & \{x = y \mid x \in \mathcal{S}[\![\, e_1 \,]\!], y \in \mathcal{S}[\![\, e_2 \,]\!]\} & \\
\mathcal{S}[\![\, e_1 \texttt{<} e_2 \,]\!]\rho & = & \{x < y \mid x \in \mathcal{S}[\![\, e_1 \,]\!], y \in \mathcal{S}[\![\, e_2 \,]\!]\} & \\
\mathcal{S}[\![\, e_1 \texttt{ and } e_2 \,]\!]\rho & = & \mathcal{S}[\![\, \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else false} \,]\!] & \\
\mathcal{S}[\![\, e_1 \texttt{ or } e_2 \,]\!]\rho & = & \mathcal{S}[\![\, \texttt{if } e_1 \texttt{ then true else } e_2 \,]\!] & \\
\mathcal{S}[\![\, \texttt{if } c \texttt{ then } t \texttt{ else } e \,]\!] & = & \left\{ \begin{array}{ll} \mathcal{S}[\![\, t \,]\!]\rho_t & \text{if } v_c = \{true\} \\ \mathcal{S}[\![\, e \,]\!]\rho_e & \text{if } v_c = \{false\} \\ \mathcal{S}[\![\, t \,]\!]\rho_t \cup \mathcal{S}[\![\, e \,]\!]\rho_e & \text{otherwise} \end{array} \right. & \\
 & & \textit{where} \quad v_c = \mathcal{S}[\![\, c \,]\!]\rho & \\
 & & \qquad\quad \rho_t = \mathcal{T}[\![\, c \,]\!]\rho & \\
 & & \qquad\quad \rho_e = \mathcal{F}[\![\, c \,]\!]\rho & \\
\mathcal{S}[\![\, \texttt{sum}(l, e) \,]\!] & = & U_{Int} & \\
 & & & \\
\mathcal{T}[\![\, \texttt{not } e \,]\!]\rho & = & \mathcal{F}[\![\, e \,]\!]\rho & \\
\mathcal{T}[\![\, e_1 \texttt{ and } e_2 \,]\!]\rho & = & \mathcal{T}[\![\, e_2 \,]\!](\mathcal{T}[\![\, e_1 \,]\!]\rho) & \\
\mathcal{T}[\![\, x = l \,]\!]\rho & = & \rho[x \mapsto \{\mathcal{L}[\![\, l \,]\!]\}] & \text{a literal} \\
\mathcal{T}[\![\, x < l \,]\!]\rho & = & \rho[x \mapsto \{v \mid v \in \rho x, v < \mathcal{L}[\![\, l \,]\!]\}] & \text{a literal} \\
\mathcal{T}[\![\, \_ \,]\!]\rho & = & \rho & \text{otherwise} \\
\mathcal{F}[\![\, \texttt{not } e \,]\!]\rho & = & \mathcal{T}[\![\, e \,]\!]\rho & \\
\mathcal{F}[\![\, e_1 \texttt{ or } e_2 \,]\!]\rho & = & \mathcal{F}[\![\, e_2 \,]\!](\mathcal{F}[\![\, e_1 \,]\!]\rho) & \\
\mathcal{F}[\![\, x = l \,]\!]\rho & = & \rho[x \mapsto \rho x \setminus \{\mathcal{L}[\![\, l \,]\!]\}] & \text{a literal} \\
\mathcal{F}[\![\, x < l \,]\!]\rho & = & \rho[x \mapsto \{v \mid v \in \rho x, v \geq \mathcal{L}[\![\, l \,]\!]\}] & \text{a literal} \\
\mathcal{F}[\![\, \_ \,]\!]\rho & = & \rho & \text{otherwise} \\
 & & & \\
\mathcal{L}[\![\, \texttt{true} \,]\!] & = & true & \\
 & & \vdots &
\end{array}
$$

Figure 6: A set based evaluator.

and then uses it as a premise for a theorem prover within each branch. We currently do not implement this since it looks like it would have little practical impact.

The set evaluator in figure 6 does not handle *void*. *Void* could be seen as an ordinary value in the set, but it can, of course, not take part of any arithmetic or other operations. *Void* also needs to be handled specially since even if we need to use the worst possible approximation (like for `sum`), i.e., the universal set, we still want to know if *void* can be part of it or not.

Using the set evaluator as a tool, it is possible to transform the program. For each subexpression the set evaluator is invoked. If it yields a singleton value, this value replaces the subexpression. If it does not yield a single value, the parts of the subexpressions are examined and transformed in the same way. After such a pass over the whole program, the constant propagation and constant folding can be rerun to take advantage of the new constants.

It might seem like a lot of work to repeatedly run the set evaluator like this. Some form of caching could be used, but practice shows that this is not essential.

3.3.1 Implementation of the value sets

The operations that the value sets must support are the usual for set operations (union, membership test, etc.). These are used in set based evaluation for handling the different language constructs.

To handle arithmetic, different operations are needed. E.g., the possible values of the expression "`x+y`" is the set $\{x + y \mid x \in v_x, y \in v_y\}$. So the operations on sets must include these cross-product like arithmetic operations.

The cardinality of the sets of values as obtained from the input data could, in the worst case, be as large as the number of legs in the flight plan. This number can range from a few hundered to 10000, but the cardinality is usually much lower than this; normally being less than 15. Unfortunately the arithmetic operations on sets cause the cardinality of the resulting sets to explode. After a few arithmetic operations the size of the set could easily exceed the memory of the computer. To be able to handle those large sets efficiently, they need to be approximated. The approximation used here is to switch from an exact set representation to an interval representation. The interval retains the smallest and largest of the elements in the original set. Arithmetic on these sets then becomes ordinary interval arithmetic.

The implementation does, in fact, switch between four representations: singleton set, exact list of elements, interval, and the universal set. The first of these are only present to improve space and time efficiency. Each of these representations also keeps track of if the set contains *void* or not.

3.4 Ad hoc transformation rules

By studying the output of the partial evaluator you can easily find a number of expressions which can be simplified. The transformations that are needed cannot be done by the constant folding, nor the set based evaluation since these only deal with actual values. The transformations of interest here are symbolic in nature.

The needed transformations range from the trivial, e.g., "`0 * x`" should be replaced by "`0`", to the not so trivial, e.g., "`count(level) where(p) > 0`" should be replaced by "`any(level, p)`".

Many of the transformation rules have side conditions that relate to *void*. E.g., "`x * 0`" can only be replaced by `0`, if `x` cannot have the value *void*, since CRL hs left-to-right evaluation so the expression is *void* if `x` is. During the transformation, these side conditions can be checked by using the set evaluator to examine the possible values of subexpressions.

Figure 7 contains a partial list of the transformations utilised. All of these have been derived ad-hoc by studying the output of the partial evaluator, but they can be verified to be correct using the semantics of the language The set of transformations could be extended, of course, but it handles many of the cases that occur in practice.

3.5 Other transformations

Some other transformations are also performed:

- Simple (e.g. a literal or a variable) definitions and definitions used only once are inlined.
- Identical definitions are coalesced.
- Unused definitions are removed.
- Simple theorem proving is used to removed rules that are covered by other rules. E.g. "`rule l.r1 = x < 5`" can be removed if there is also a rule "`rule l.r1 = x < 0`".

4 Implementation

The partial evaluator reads a dump of the abstract syntax tree of the input program and generates a new rule set in concrete syntax. The dump of the abstract syntax tree is generated by the ordinary CRL compiler. The output of the partial evaluator is a quite readable program (this was one of the requirements of it), some of the comments are even preserved.

The way the system is used without partial evaluation follows these steps:

`if true then` t `else` e	$\Rightarrow$	t	
`if false then` t `else` e	$\Rightarrow$	e	
`if` c `then true else false`	$\Rightarrow$	c	
`if` c `then false else true`	$\Rightarrow$	`not` c	
`if not` c `then` t `else` e	$\Rightarrow$	`if` c `then` e `else` t	
`if` c `then` t `else if` c `then` t' `else` e	$\Rightarrow$	`if` c `then` t `else` e	
(`if` c `then` t `else` e)+x	$\Rightarrow$	`if` c `then` t+x `else` e+x	x variable
x+(`if` c `then` t `else` e)	$\Rightarrow$	`if` c `then` x+t `else` x+e	x variable
		$\vdots$	
e=`true`	$\Rightarrow$	e	
e=`false`	$\Rightarrow$	`not` e	
e+`0`	$\Rightarrow$	e	
`0`+e	$\Rightarrow$	e	
e*`1`	$\Rightarrow$	e	
`1`*e	$\Rightarrow$	e	
e*`0`	$\Rightarrow$	`0`	if $void \notin \mathcal{S}[\![e]\!]\rho$
`0`*e	$\Rightarrow$	`0`	
$e-e$	$\Rightarrow$	`0`	if $void \notin \mathcal{S}[\![e]\!]\rho$
		$\vdots$	
e `and` e	$\Rightarrow$	e	
e `or` e	$\Rightarrow$	e	
`not` (`not` e)	$\Rightarrow$	e	
`not` (e_1 `and` e_2)	$\Rightarrow$	`not` e_1 `or not` e_2	
`not` (e_1 `or` e_2)	$\Rightarrow$	`not` e_1 `and not` e_2	
`not all`(l, e)	$\Rightarrow$	`any`(l, `not` e)	
`not any`(l, e)	$\Rightarrow$	`all`(l, `not` e)	
`sum`(l, `0`)	$\Rightarrow$	`0`	
`all`(l, `true`)	$\Rightarrow$	`true`	
`any`(l, `false`)	$\Rightarrow$	`false`	
`any`(l, e_1) `and any`(l, e_2)	$\Rightarrow$	`any`(l, e_1 `and` e_2)	
`count`(l)`where`(e)`> 0`	$\Rightarrow$	`any`(l, e)	

Figure 7: A few of the symbolic transformation rules.

- Compile rule set using the CRL compiler. The CRL compiler translates the rule language to C which is then compiled with a standard C compiler. For the user this step looks like one operation.
- Use the compiled rule set to solve different problems. Different problems may have different parameters, different external table, and different leg sets as input. The compiled rule set is dynamically linked with the runtime system for each run.

It is important to have the system look exactly the same when the partial evaluator is in use. The steps looks the same to the user, but they are different internally.

- Compilation of the rule set will invoke just the CRL compiler which will collect the used modules together and produce a dump of the abstract syntax tree.
- To use a "compiled" rule set the following happens:
  - The partial evaluator is run with the dumped rule set, the current parameters, the external tables, and the leg set as input. These are the same values as would be used by the runtime system had the partial evaluator not been used.
  - The resulting rule set is then compiled with the CRL compiler (which uses the C compiler), and then used by the runtime system just as above.

The partial evaluator consists of about 7000 lines of Haskell, [Hud92], code. Early in the development the Hugs system, [Jon96a, Jon96b], was used, but after an initial period the HBC compiler, [Aug93], was used exclusively. Heap profiling [RW93] and time profiling was used to improve the performance of the program.

5 Practical evaluation and conclusions

The partial evaluator has been used on most of the rule sets written in CRL. Table 1 presented the figures for a Lufthansa problem. Lufthansa has the biggest rule set of all the airlines.

As can be seen from the table the running time decreases to about half. When tested on a larger variety of problems the running time varies between 30% and 65 % of the original time, with 50% being a typical figure. When you consider that these runs most often takes several hours this is a quite a gain. The total time for processing (rule compilation + partial evaluation + C compilation) the rule set has also decreased. The rule compiler generates a

	Before PE	After PE
Rules	88	90
Leg definitions	209	68
RTD definitions	391	106
CRR definitions	192	22
Total rule size	461 kbyte	176 kbyte
C code generated	2.17 Mbyte	0.99 Mbyte
Time for PE	-	3 min
Time for C compilation	30 min	10 min
Running time	55 min	25 min

Table 1: Summary of partial evaluation results for a Lufthansa rule set partially evaluated for the Lufthansa Express (a few aircraft types flying within Europe) problem for a captain. The run time is for solving a small subset of a weeks scheduling.

huge C program, which takes very long to compile.[3] The number of definitions is reduced because unused definitions are removed (a few) simple definitions are inlined (many). Many definitions becomes simple after the transformations. The reason the number of rules has increased is the cloning of leg level definitions to handle deadheads. Without this cloning the number of rules would also have been reduced.

The running time of the partial evaluator has not been a problem and is not remarkably long when one considers that, in the example in the table, the in input program is almost five hundred kbytes, the external tables another hundred kbytes, and the given attributes for the legs in the flight plan is about five hundred kbytes.

Even if the running time of the transformed program decreased by a fair amount this is, surprisingly, not always where the biggest gain in speed was made. For a particularly difficult planning problem the output from the partial evaluator, which is human readable, was inspected by an expert planner/rule-writer. Since the rule set had decreased so much in size he was able to grasp what the rules meant and make some manual adjustments to them. These adjustments made the program run an order of magnitude faster. These changes could, of course, have been made in the original program, but studying and comprehending it would have been a daunting task.

The partial evaluator is now part of the Carmen system and is delivered to several customers that use it in their daily work.

[3] Actually, it has to be broken into smaller pieces since most C compiler cannot compile files that are several megabytes.

6 Acknowledgements

I would like to thank the people at Carmen Systems, especially Tommy Bohlin, who answered my naïve questions about airline scheduling problems. I would also like to thank Jessica Twitchell for proof reading and improving the English of this paper.

References

[AGPT91] R. Anbil, E. Gelman, B. Patty, and R. Tanga. Recent Advances in Crew-Pairing Optimization at American Airlines. *Interfaces*, 21(1):62–74, 1991.

[AHKW97] E. Andersson, E. Housos, N. Kohl, and D. Wedelin. Crew Pairing Optimization. In *OR in Airline Industry*. Kluwer Academic Press, 1997.

[Aug93] L. Augustsson. *HBC User's Manual*. Programming Methodology Group, Department of Computer Sciences, Chalmers, S–412 96 Göteborg, Sweden, 1993. Distributed with the HBC compiler.

[Boh90] Tommy Bohlin. The Carmen Rule Language. Technical report, Carmen Systems AB, 1990.

[Hud92] Paul Hudak et al. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.

[Jon96a] Mark P. Jones. Hugs 1.3 user manual. Technical Report NOTTCS-TR-96-2, Department of Computer Science, University of Nottingham, August 1996.

[Jon96b] Mark P. Jones. The Hugs distribution. Currently available from `http://www.cs.nott.ac.uk/Department/Staff/mpj/hugs.html`, 1996.

[RW93] Colin Runciman and David Wakeling. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–245, April 1993.

[TE96] E. Housos T. Elmroth. Automatic Subproblem Optimisation for Airline Crew Scheduling. *Interfaces(to appear)*, 1996.

[Wed95] D. Wedelin. An algorith for large scale 0–1 integer programming with application to airline crew scheduling. *Annals of Operations Research*, 57:283–301, 1995.