INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67, D–80538 München

Ludwig-Maximilians-Universität München

The Functional Rent Advisor

Slim Abdennadher **Tim Geisler** **Sven Panne**

Submitted for publication
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-1998-12, August 1998

The Functional Rent Advisor

Slim Abdennadher, Tim Geisler, and Sven Panne

Institut für Informatik, Universität München, Oettingenstr. 67, D-80538 München
{abdennad|geisler|panne}@informatik.uni-muenchen.de
http://www.pms.informatik.uni-muenchen.de/software/fra/

Abstract. The Functional Rent Advisor (FRA) is an intelligent Internet application that allows calculating an estimate of the fair rent for a flat in Munich. The FRA is a re-implementation of the Munich Rent Advisor [FA96] using the high-level purely functional language Haskell. This paper shows that Haskell is appropriate to realize non-trivial Internet applications. Simplicity, flexibility and rapid prototyping were the advantages of using Haskell.

1 Introduction

The Munich Rent Advisor (MRA) [FA96,FA97] is the electronic version of the "Mietspiegel"(MS) which is published regularly by the City of Munich. The MS basically contains a verbal description of an expert system, which is derived from a statistical model compiled from sample data using statistical methods such as regression analysis [All94]. It allows to calculate the estimated fair rent for flats. The results of these calculations are typically used in civil court cases. The calculations are based on size, age and location of the flat and a series of detailed questions about the flat and the house it is in. Some of these questions are hard to answer. However, in order to be able to calculate the rent estimate by hand, all questions must be answered. The MRA extended the functionality and applicability of the MS so that the user need not answer all questions of the form. The MRA now can be used not only for calculating the estimated fair rent of a flat but also for helping house-hunters which have a vague idea of the kind of the flat they plan to rent and are interested in the rent they have to meet.

Equipped with pencil, paper and calculator, one may need a weekend to figure out the estimated rent. Usually, the calculation is performed by hand in about half on hour by an expert from the City of Munich or from one of the renter's associations. The MRA brought the advising time down to a few minutes that the user needs to fill in the form.

The original MRA [FA96] is implemented in Prolog, using Constraint Handling Rules [Frü98] a high-level language extension to write constraint systems. In that paper, it is mentioned that the MRA is quite atypical from a constraint logic programming point of view, since it is not concerned with the constraint-pruned search for a solution of NP-hard problems, but performing a calculation in the presence of partial information. One major contribution of constraints in [FA96] is to problem modeling which is one of the most important features of constraint programming [Wal96].

In this paper we present a re-implementation of the MRA using the high-level purely functional language Haskell [PH997]. We are using the same approach as presented in [FA96], i. e., partial information is represented by intervals. We show that we can simply implement the needed operations for interval arithmetic using Haskell to express the imprecision due to the statistical approach and incompleteness due to partial user answers. The tables, rules and formulas of the "Mietspiegel" can be specified in a declarative way in Haskell, too. Our high-level approach implies that the program can be easily maintained and modified. This is crucial, since every city and every new version of the "Mietspiegel" comes with different tables and rules.

Although there is already an existing implementation in Prolog, we re-implemented the MRA in Haskell for several reasons: Being a strongly typed language with an advanced type system [WB89] and powerful abstraction facilities due to modules and higher-order functions, Haskell supports software engineering principles better than Prolog does. The correspondence between the "paper version" and the code is much closer than in its Prolog precursor. This fact can mainly be attributed to the more exact modeling of the problem, e. g., different kinds of questions are represented by different data types. Another reason for this switch of the implementation language was the availability of a convenient library for HTML generation and CGI scripting [Mei97]. This relieves one from the error-prone tasks of generating correct HTML and handling of the HTTP protocol.

The paper is organized as follows. The next section introduces the "Mietspiegel". Section 3 describes the Web front end and section 4 presents the implementation. We conclude with a summary and directions for future work.

2 The "Mietspiegel"

The "Mietspiegel" (MS) is published every other year by the housing group of the department for social issues of the city administration of Munich after negotiations with renter's and landlord's associations and lawyers. The "Infratest Sozialforschung" in Munich together with the "Institut Wohnen und Umwelt" (institute for housing and environment) in Darmstadt conducted about 7000 interviews with a 27 page questionnaire to obtain the sample data. About a third of the interviews were useful to build the statistical model at the Department of Statistics of the Ludwig-Maximilians-Universität München [All94].

The scheme for calculating the rent estimate is roughly as follows:

$$\begin{aligned} \textit{Estimated Rent} = {} & \textit{Size} * \textit{Basic Rent per Squaremeter} \\ & * (\textit{Sum of Deviations as Percentage} + 100) * 0.01 \\ & * (\textit{Imprecision Deviation Percentage} + 100) * 0.01 \\ & + \textit{Fixed Costs} \end{aligned}$$

The calculation starts with the average rent per square meter taken from a table with about 200 entries. From this anti-monotonic function (the bigger the flat, the lower this cost) the average rent is calculated.

The deviations from the average rent are computed from the answers regarding the size, location, features of the flat, as well as age and state of the house. There are five yes/no questions about features of the house concerning e. g., number of floors, optical impression, lift, and eleven yes-no questions about features of the flat concerning e. g., central heating, separate shower, or dish-washer. The answers to these questions, sometimes combined with the age of the house, yield the deviations from the average rent.

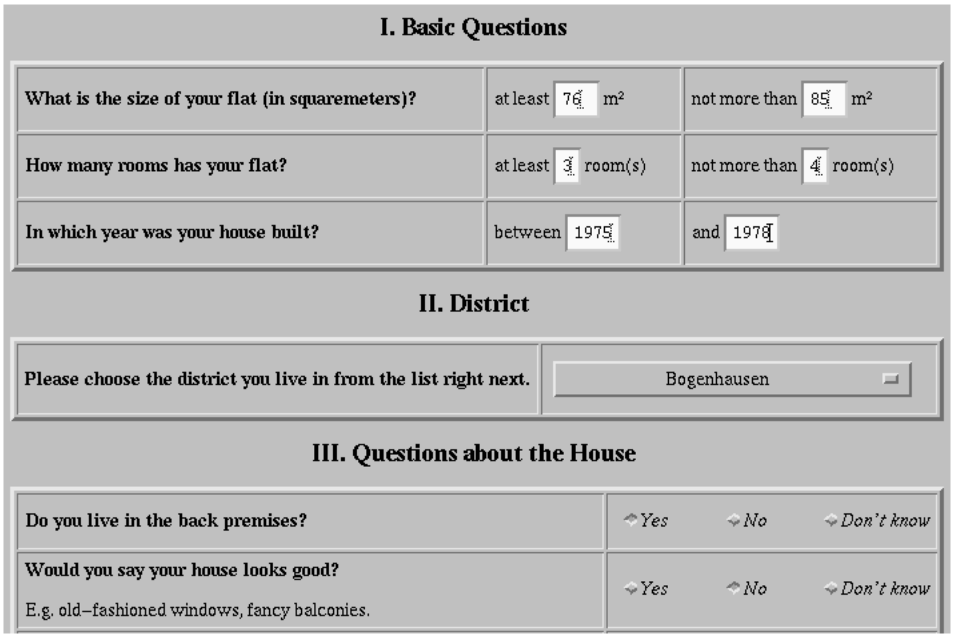
The main deviation comes from the age of the house, the number of rooms in the flat and if it has a large balcony. The overall deviation may be up to $\pm 60\%$.

For the MS, the complex data sets derived from the interview have been reduced and simplified so that an average person could calculate the estimated rent. As we have pointed out in the introduction, the MS is still too complicated to be used by everybody. In addition, it ignores the inherent imprecision of the statistical model. The imprecision is basically the standard deviation obtained in the statistical model. Therefore it is higher for rare kinds of flats (very small, big or very old, new etc.). On average, the imprecision deviation amounts to about $\pm 10\%$.

Finally one has to add fixed costs, e. g., taxes, fees for rubbish dump, house cleaning, cable TV and other service charges. Part of them may be included in the rent, part of them not, part of them may not apply. There are currently 16 items on the list of fixed costs. Usually, the user will just ignore this section and thus a range from minimal to maximal fixed costs will be added to the estimated rent. Detailed answers (and thus detailed results) only make sense if the user wants to go to court because he is overcharged with the fixed costs.

3 The Web Front End

We provide a Web front end as a user interface to the electronic version of the "Mietspiegel" – *The Functional Rent Advisor (FRA)* by implementing the whole application as a CGI-bin. Choosing this kind of user interface has several advantages. The user interface is easy to implement with high-level languages like Haskell which do not yet have commonly accepted and portable graphical user interface libraries. Furthermore, the user interface is very portable: it can be run (i. e., displayed) on nearly any Web browser. HTML provides some standardized internationalization features. The FRA selects its language from the preferences specified by the user in her browser configuration. The most important advantage for research prototypes is the ease of distribution: the only administrative labour is to install the CGI-bin on a Web server. All relevant information for calculating an estimated rent will be collected in the questionnaire (Fig. 1). FRA users need to fill in only what they know and what they care about. All answers are optional. There are only four questions requiring numeric inputs, where it is possible to give a range (editable fields) and one question about location requiring a search in a list of districts (pull-down menu). The remaining questions are



I. Basic Questions

What is the size of your flat (in squaremeters)?	at least 76 m²	not more than 85 m²
How many rooms has your flat?	at least 3 room(s)	not more than 4 room(s)
In which year was your house built?	between 1975	and 1978

II. District

Please choose the district you live in from the list right next.	Bogenhausen

III. Questions about the House

Do you live in the back premises?	Yes	No	Don't know
Would you say your house looks good? E.g. old–fashioned windows, fancy balconies.	Yes	No	Don't know

Fig. 1. A fragment of the questionnaire.

multiple choices, where the only possible answers are *Yes*, *No*, and, in addition, *Don't know/care* (buttons). Optional detailed questions are provided to calculate the fixed costs where numeric input can be given. This form is divided in five sections: basic questions, a question about the district the house is in, questions about the house, questions about the flat itself, and questions about the fixed costs of the flat. These questions were sorted by importance of the answer to estimate the rent. Questions at the beginning of a section have more influence on the result than questions at the end of a section. After submitting the form, the user gets the estimated rent, including some information about the deviations (Fig. 2).

4 Implementation

Regarding the implementation, the main observation is that the estimated rent is a *function* of the information given by the user. This suggests using a CGI-bin, which can be seen as a function map-

Type	Result in DM
Rent	between 1276.29 and 1871.55
Rent without 'Nebenkosten'	between 1076.29 and 1671.55
'Nebenkosten'	200.00

Even if you have answered all questions, there will still be some imprecision due to the statistical model used.

Changes for	Percentage
Number of rooms and age of building	between –3.0 and 10.0
Living area	15.0
Renovations	10.0
Special windows	between –6.0 and –0.0
Design of the flat	10.0
Bathroom	13.0
Kitchen and bathroom	19.0

Fig. 2. A fragment of the resulting calculation.

ping an environment (key/value pairs) to a MIME content (HTML in our case). Only a single CGI-bin is used, which is called two times: First the questionnaire is generated and after submission of the user's information the output page is calculated. The two cases are distinguished by a special key in the CGI-bin's environment. This approach ensures consistency between the questionnaire and the actual calculation and makes the installation of the FRA easier. Furthermore, other front-ends for the FRA are possible. They can connect to the calculation part by submitting an URL including the data about the flat.

4.1 CGI Scripting

Writing an CGI application from scratch is an error-prone and rather low-level task. Part of the HTTP protocol has to be implemented, URLs have to be parsed and HTML has to be generated. Part of the popularity of Perl for writing CGI-bins stems from the fact that a library for this task was available at an early point. More recently, libraries for other languages became available, e. g., PiLLoW/CIAO [CHV96] for Prolog. We use a modified version of Erik Meijer's CGI-library [Mei97] for Haskell.

Using such a library, even for a static HTML page, has several advantages: The low-level communication is hidden and the programmer can represent complex HTML objects, e. g., tables, in a structured way rather than by flat byte streams. Some errors which are otherwise easily made, like unbalanced HTML elements, are impossible this way. Furthermore, abstraction facilities of Haskell can be employed to build higher-level entities, like a question for a numeric interval or groups of different questions. But even with current libraries, incorrect HTML pages can be constructed because of two weaknesses:

– HTML elements can be incorrectly nested, e. g., anchors within anchors.
– Required attributes for HTML elements can be omitted, unused attributes can be given, or attribute values can have a wrong format.

Using a strongly typed language like Haskell, one should be able to do better. It would be nice if the errors above would result in static type errors at compilation time and not browser errors at runtime.

Although the first problem looks easy, it is not: HTML is full of rules like: "You can nest anything within x, *except for* y." Even if it was possible to rewrite the HTML grammar without these rules, it would probably result in something incomprehensible to reader. This is similar to the dangling-else problem in lots of programming languages where the grammar itself is ambiguous, but an additional rule is given: an "else" belongs to the last "if".

The second problem boils down to a clean modeling of attributes and *non-disjoint* sets of attributes (e. g., core attributes or attributes accepted by anchors in HTML). Very similar problems are common for a lot of libraries, a prominent example being the graphics API OpenGL^{tm} [WND97].

Both are open problems, for which solutions are not yet known to the authors, so more research in these directions is needed.

4.2 Processing the HTML Form Values

The HTML input form contains several different input elements for questions about features of the house or flat. Each of them has a unique feature name and some default value. There are certain types of values: intervals of numbers, intervals of truth values and strings.

The most difficult and unsafe part of the rent advisor application is the transformation of the feature's names and values to and from the textual representation used in the HTML input form and answer environment.

These types of feature values can be distinguished in Haskell by defining a separate algebraic data type for each type of feature. The feature names are now constructors for the respective feature type. Together with the representation of questions as different algebraic data types this approach enables some compile time checking and leads to a safer transformation of the feature's names and values to and from their textual representation.

In the rent estimate calculation function, locally defined and partially applied value lookup functions (one for each feature type; named `bVal` for booleans and `fVal` for floats in the sequel) map the feature names to their values.

4.3 Interval Arithmetics

The implementation of interval arithmetics in Haskell is straightforward. We define an algebraic data type `Interval` with suitable instances of the type classes `Num` and `Fractional`. Furthermore, we define constructor (...) and selector (`low`, `high`) functions as well as an infix function `|-|` which cuts an interval such that it fits into another interval, and an infix function `-=-` which tests two intervals for overlap.

```
data Interval a = I a a

Ord a => a -> a -> Interval a
x ... y | x <= y    = I x y
        | otherwise = I y x

low, high :: Interval a -> a
low  (I l _) = l
high (I _ h) = h

instance (Num a, Ord a) => Num (Interval a) where
   I l1 h1  +  I l2 h2 = I (l1+l2) (h1+h2)
   I l1 h1  *  I l2 h2 = let bounds = [l1*l2, l1*h2, h1*l2, h1*h2]
                         in  I (minimum bounds) (maximum bounds)
   negate (I l h)      = I (negate h) (negate l)
   fromInteger i       = let fi = fromInteger i in I fi fi
```

```
instance (Fractional a, Ord a) => Fractional (Interval a) where
   fromRational r      = let fr = fromRational r in I fr fr

(-=-) :: (Ord a) => Interval a -> Interval a -> Bool
I l1 h1  -=-  I l2 h2  = l1<=l2 && l2<=h1 || l2<=l1 && l1<=h2

(|-|) :: (Ord a) => Interval a -> Interval a -> Interval a
I l h  |-|  I cl ch | l > ch    = I ch ch
                    | h < cl    = I cl cl
                    | otherwise = I (max l cl) (min h ch)
```

By using Haskell's type classes and overloading, the formula for calculating the rent estimate using intervals can be written as if no intervals were used.

The presence or absence of several of the house's or flat's features results in positive or negative deviations from the estimated rent. Even yes-no questions about the presence of such features may remain unanswered by the user. Therefore, the answers should be represented as intervals of truth values.

The Prolog implementation of the rent advisor uses the integer intervals $[0, 0]$, $[1, 1]$, and $[0, 1]$ to represent false, true, and an unknown truth value, respectively. The conditional use of a deviation is then computed by simply multiplying the condition interval and the deviation interval. It is obvious that this lack of type information can be very error prone. For example, the addition of two intervals representing true $[1, 1]+[1, 1]$ would result in the interval $[2, 2]$ which is not an interval of truth values.

The cleaner solution is to directly use the type `Interval Bool` to represent intervals of truth values. By conservatively extending Haskell's predefined type class hierarchy, it suffices to make the type `Bool` an instance of the type class `Num`. As a convenient user interface, we provide the functions `false`, `true`, and `unknown` to construct intervals of truth values and the functions `isFalse`, `isTrue`, and `isUnknown` to test for certain intervals of truth values (their trivial implementation is not given in this paper).

```
instance Num Bool where
   (+)         = (||)
   (*)         = (&&)
   negate      = not
   fromInteger = (/=0)
```

We furthermore need functions to compare intervals, i. e., to define an ordering relation on intervals. It is not a good idea to define an instance of `Ord` for that purpose, because a lot of functions from various Haskell libraries assume that the order `<=` is total. For the rent advisor application, it is more appropriate to define comparison functions that return intervals of truth values.

```
(<:), (<=:), (>:), (>=:) ::
  (Num a, Ord a) => Interval a -> Interval a -> Interval Bool
I l1 h1 <:  I l2 h2 | h1 <  l2  = true
                    | l1 >  h2  = false
                    | otherwise = unknown
I l1 h1 <=: I l2 h2 | h1 <= l2  = true
                    | l1 >= h2  = false
                    | otherwise = unknown
i1 >:  i2                       = i2 <:  i1
i1 >=: i2                       = i2 <=: i1

(Num a, Ord a) => Interval Bool -> Interval a -> Interval a
c ==> i | isTrue  c = i
        | isFalse c = 0
        | otherwise = (I 0 1) * i
```

In the rent advisor application, conditionalized values occur: if a condition is true, then some given value is used; if it is false, 0 is used. The function `==>` lifts these conditionalized values to intervals. Note that these conditionalized interval expressions resemble rules in a deductive language.

4.4 Tables and Rules

The "Mietspiegel" contains several one- or two-dimensional tables indexed by intervals that relate features of the flat to percentual changes of the estimated rent. For example, the rent depends on the age of the flat and its number of rooms. The table to describe this function as found in the MS is (translated from *Tabelle 3* in [DO97]):

Year of construction	1 room	2–3 rooms	≥ 4 rooms
⋮	⋮	⋮	⋮
1966–1977	-3.5	-2.0	-3.0
1978–1983	2.0	10.0	3.0
1984–1986	6.0	18.0	7.0
⋮	⋮	⋮	⋮

In Haskell, such a multi-dimensional table is represented declaratively as a list of tuples consisting of one or more keys (which are intervals) and a value:

```
tConstructionYearRooms = [ ...   -- ommitted
                         , (1966...1977,      1, -3.5)
                         , (1966...1977, 2...3, -2)
                         , (1966...1977, 4...9, -3)
                         , (1978...1983,      1,  2)
                         , (1978...1983, 2...3, 10)
                         , (1978...1983, 4...9,  3)
                         , (1984...1986,      1,  6)
                         , (1984...1986, 2...3, 18)
                         , (1984...1986, 4...9,  7)
                         , ...   -- ommitted
                         ]
```

A query to a table with interval-typed keys returns an interval of values, which is the convex hull of all values with corresponding key intervals that overlap the query intervals. This query operations can be implemented in a few lines of Haskell using list comprehensions.

A table (`tBalcony`) relates the age of the flat to a percentual change of the estimated rent, provided the flat has a large balcony. This conditional table lookup can be written as follows:

```
balcony = bVal LargeBalcony ==> tLookup1 tBalcony cy
```

Another table describing the deviation for some features in the flat is represented as a function `dev` mapping features to deviations.

Furthermore, the "Mietspiegel" contains some rules. For example, the extra imprecision deviation is determined by the following rule (translated from [DO97]):

> [...] For flats with a size below $30\,m^2$ or above $100\,m^2$ the imprecision deviation is to be extended by $\pm 2\%$, as well as for flats built after 1984.

This can be extended to consider imprecise information and written in Haskell declaratively as follows (the interval `cy` is the flat's construction year and the interval `size` is the flat's size in square meters). Notice that in this context the overloaded function `+` (resp. `*`) is the logical disjunction (resp. conjunction) on intervals of truth values.

```
imprecDev = size <: 30 + size >: 100 + cy >=: 1984 ==> -2.0...2.0
```

There is a further deviation for back premises built before 1949:

```
backPrem = bVal BackPrem * (cy <: 1949) ==> dev BackPrem
```

The maximal deviations for certain features of the flat are limited (translated from [DO97]):

> The total deviation for extra facilities in bathrooms must not exceed 13%. The total deviation for extra facilities in kitchen and bathrooms must not exceed 19%.

The total deviation of these features with respect to these limits can be described very conveniently in Haskell:

```
sanDev  = sum (map pDev [BathLux, BathExtras, BathSnd]) |-| 0...13
ksanDev = (sanDev + pDev KitchenExtras) |-| 0...19
```

Note that all the above definitions are well-typed. Haskell's type inference permits the user to omit the type signatures and thus retain a short and readable specification, but helps enormously to detect some errors in the specification at compile time.

4.5 Rent Estimate Calculation

As mentioned in section 2, the actual rent estimate calculation is just a single, albeit complex, expression. This expression maps imprecise numbers and truth values represented as intervals to one output interval, using the functions presented above as building blocks.

Note that the rent estimate calculation is a functional problem, which can directly be represented in a functional programming language. A logic programming language like Prolog, which was used in the original implementation of the MRA [FA96], is not equally handy. The relational notion of standard Prolog does not allow nested expressions and requires the introduction of many auxiliary variables, which can be very error-prone.

In order to provide the user with some explanation of the calculation, the single expression is split into several smaller subexpressions. The values of the whole expression and its subexpressions are subsequently used as input to the generation of the answer page.

The central part of the rent estimate calculation is given below. Due to its declarativity, this code is very short – about one page of Haskell code – and is therefore easy to understand and maintain. It is even smaller than the written specification of the “Mietspiegel”.

```
let size      = fVal Size
    cy        = fVal ConstructionYear
    rooms     = fVal Rooms
    ⋮
    basicRent = tLookup1 tBasicRent size
    cyR       = tLookup2 tConstructionYearRooms cy rooms
    ⋮
    backPrem  = bVal BackPrem * (cy <: 1949) ==> dev BackPrem
    balcony   = bVal LrgBalcony ==> tLookup1 tBalcony cy
    ⋮
    percDev   = cyR + backPrem + balcony +
                ⋮
    sanDev    = sum (map pDev [BathLux,BathExtras,BathSnd]) |-| 0...13
    ksanDev   = (sanDev + pDev KitchenExtras) |-| 0...19
    imprecDev = size <: 30 + size >: 100 + cy >=: 1984 ==> -2.0...2.0

    estRent   = basicRent *
                inPercent ((percDev+ksanDev) |-| (-70...95)) *
                inPercent generalImprecDev *
                inPercent imprecDev
in page ... -- Answer page generation
where
   bVal   = val booleanValueAssocs
   fVal   = val floatValueAssocs
   pDev k =        bVal k  ==> dev k
   nDev k = negate (bVal k) ==> dev k
   ... -- Some more local functions
```

5 Conclusion and Future Work

In this paper, we have argued that declarative languages like Haskell are a good vehicle for implementing practical applications. With Haskell as a specification language, one can obtain a transparent representation of the (relationships underlying the) problem. Starting from the Prolog implementation of the MS, it took only one man week to re-implement the whole program.

The Functional Rent Advisor also identified the need for lightweight declarative implementations that can run over the Internet.

Our very high-level approach implies that the program can be easily maintained and modified. This is crucial, since every city and every new version of the "Mietspiegel" comes with different tables and rules. The "Mietspiegel" in Munich seems to be the most complex one in Germany. Unfortunately, the MS in other German cities are not instances of it. Nevertheless, if a rent advisor were to be implemented for another city, a large amount of our code could be recycled. One direction for future work is to generalize our framework in order to be able to create electronic versions for all German cities.

The HTML typing issues discussed in section 4.1 have to be investigated closer, including the automatic generation of HTML libraries (for constructing, parsing, and pretty-printing) from the associated *document type descriptions* (DTDs). This should be possible, as all necessary information for this task is present in the DTDs in a formalized way. Automation in this area would greatly simplify the task of generating correct libraries for different instances of SGML, e. g., the different HTML versions.

Acknowledgments

We thank Norbert Eisinger for encouraging us to write this paper, Thom Frühwirth for the idea of actually implementing the MS, Martin Josko for maintaining our Web server and for advice on CGI-bins, and Heribert Schütz for his steadfast belief in Haskell as an excellent prototyping language.

References

[All94] R. Alles. Gutachten zur Erstellung des Mietspiegels für München '94. Sozialreferat der Stadt München – Amt für Wohnungswesen et. al., München, Germany, 1994. In German.

[CHV96] Daniel Cabeza, Manuel Hermenegildo, and Sacha Varma. The PiLLoW/CIAO library for Internet/WWW programming using computational logic systems. In Tarau et al. [TDDBH96].

[DO97] DM-Online. DM-Immobilien, Mietspiegel. http://www.dm-online.de/ib/, 1997. In German.

[FA96] Thom Frühwirth and Slim Abdennadher. The Munich Rent Advisor. In Tarau et al. [TDDBH96].

[FA97] Thom Frühwirth and Slim Abdennadher. Der Mietspiegel im Internet, Ein Fall für Constraint-Logikprogrammierung. *KI – Künstliche Intelligenz, Themenheft Constraints*, April 1997. In German.

[Frü98] Thom Frühwirth. Theory and practice of constraint handling rules,. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3), 1998.

[Mei97] Erik Meijer. CGI programming. http://www.cse.ogi.edu/~erik/, 1997.

[PH997] The Haskell 1.4 Report. http://www.haskell.org/report/, April 1997. John Peterson and Kevin Hammond (eds.).

[TDDBH96] Paul Tarau, Andrew Davison, Koen De Bosschere, and Manuel Hermenegildo, editors. *1st Workshop on Logic Programming Tools for INTERNET Applications, JICSLP '96*, Bonn, Germany, 1996.

[Wal96] Marc Wallace. Practical applications of constraint programming. *Constraints Journal*, 1(1,2), September 1996.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Principles of Programming Languages*, January 1989.

[WND97] Mason Woo, Jackie Neider, and Tom Davis. *The OpenGL Programming Guide*. Addison-Wesley, 1997.