Bootstrapping a Language Workbench

A Light-weight Approach to Language Oriented Programming

Moss Prescott and Jeremy Siek
University of Colorado, Boulder
{moss.prescott,jeremy.siek}@colorado.edu

Abstract

[TODO: Tried to start here but it was a disaster so instead worked on reducing the thesis to a compressed outline instead.]

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Syntax

Keywords Syntax extension, Domain Specific Languages, Abstract syntax

1. Introduction

intro: position rel. to other work. summarize entire paper.; the most important connected ideas

Language Workbench

- AST-based, edit via projection, execute via generation

- workbench provided many of the services of an IDE as common platform

- implementations are multi-man-decade efforts

LISP approach

- s-expressions (code and data), small kernel language, simple execution model

- extension via reduction

- limitations of s-expressions (e.g. hygiene)

define what the thing is, what the methodology is (lang. oriented meta-prog?)

thesis:

- methodology enables system to be an order of mag. smaller than same thing using previous approaches

– wider range of extensions

– also simpler/smaller

– prototype

– case studies that illustrate

- lessons learned

1.1 Roadmap

Section 2 illustrates the vision with examples using our prototype system. Section 3 presents an overview of the system. In Section 4 we describe how programs are represented and how transformations are defined. Our new language for programs and mathematical notation is presented in Section 5. In Section 6, we evaluate the success of the system by comparison with some existing tools. Related work is discussed in Section 7, and Section 8 concludes.

2. Language-oriented Meta-programming

The Language Oriented[10] approach essentially consists of three steps:

1. Define a language which accurately and conveniently expresses the program you want to write.
2. Implement the program in the new language.
3. Implement the new language using existing tools.

2.1 Defining the Core Language via Syntax Extension

The core language is built via syntax extension on top of the kernel language, so its definitions can serve as a test of the suitability of Lorax's **grammar** facilities for building these kinds of extensions. This section compares one of the declarations from Lorax's core language with the corresponding elements from Lisp and free-form languages in terms the effort invested and the benefit accrued.

Several core language nodes provide support for using the primitive cons-list values of the Clojure platform, which are one of the basic tools for organizing data in any Lisp. These extensions employ a handful of Clojure primitives to expose the native list values of the platform, and the rest of the syntax is built around them.

2.1.1 List Comprehension in Lorax

Figure 1 shows the declarations of the two primitive constructs for working with with lists (**cons** and **match-cons**), and a list comprehension (**for**) node constructed from them. For the present purpose, the first two nodes can be regarded as part of the base language, and the list comprehension syntax as an extension to be added to that language.

The *expand* reduction for the **for** node evaluates its **seq** child, and then applies a recursive function to it. This function attempts to match its argument as a non-empty list (using **match-cons**). If so, the **x** child is bound to the first value, and **expr** is evaluated and then the function is applied to the rest of the list. Note that one of the bindings (**x**) was supplied by the programmer, and the other (xs) is local to the reduction.

2.1.2 Concrete Syntax

In addition to this definition of semantics, Lorax's declaration defines a new syntax for the comprehension, which is intended to be familiar from both the mathematical notation of set theory and the comparable construct in Haskell. In Lorax, the reduction for this syntax is entirely trivial, simply giving the arrangement of the three child nodes and the symbols to be placed between them. The effort involved to provide such a simple reduction is essentially nil. For

[TODO: add []'s]

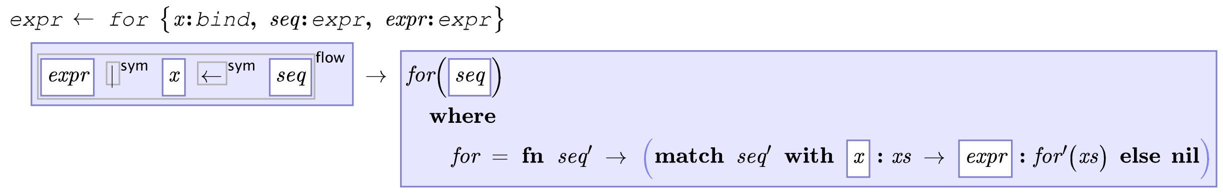

expr ← for {x:bind, seq:expr, expr:expr}
expr
sym
x
←
sym
seq
flow
→
for(seq)
where
for = fn seq′ → (match seq′ with x : xs → expr : for′(xs) else nil)

Figure 1. Defining a syntax for list comprehensions.

the programmer using the extended language, this syntax gives list comprehensions a distinct appearance.

2.1.3 Evaluation

Lorax's facilities for declaring new syntax and specifying semantics and concrete syntax allow new syntax to be introduced with an effort that compares well to Lisp's macro facility. However, both the declaration of the new syntax and its use are significantly more readable than the corresponding Lisp programs, or even the syntax from Haskell which is designed for just this purpose. Meanwhile, the effort involved is much less than the effort would be to add such a construct to a language such as Haskell.

2.2 Introducing a New Runtime Value

The preceding section showed how the facilities of the platform can be exposed and wrapped in a novel syntax. This syntax helps the programmer to understand the program, but as soon as the program is reduced to the kernel language for evaluation, the syntax is gone and only the primitive values remain. A more ambitious goal for syntax extension is to augment the language with a new kind of runtime value. This allows programs to operate on a new kind of data, and allows the programmer to see results of computation in the natural form. The next example shows how a new kind of value can be introduced, and how it supports writing a program in a much more natural and comprehensible way.

2.2.1 Enumerating the Positive Rationals

In a delightful Functional Pearl [3], Gibbons et al. present Haskell programs which generate the infinite series of all positive rational numbers. They begin with the idea of traversing the infinite matrix $a_{ij} = i/j$ which contains every positive ratio, but also contains many equivalent, unreduced ratios (e.g. ${}^1\!/_2$, ${}^2\!/_4$, . . .).

The authors show that a series containing all positive rationals in reduced form, without duplicates, is obtained by iterating the following function beginning with $x = 1$:[1]

$$x' = 1/(\lfloor x \rfloor + 1 - \{x\})$$

This is a surprisingly simple formula, but it is somewhat computationally undesirable in that calculating $\lfloor x \rfloor$ and $\{x\}$ involves division.

Interestingly, this formula can be implemented using only "a constant number of arbitrary-precision integer additions and subtractions, but no divisions or multiplications" by choosing a different representation for ratios. It happens that the five necessary operations—reciprocal, floor, addition of an integer, negate, and fractional part—can all be efficiently performed on ratios represented as *regular continued fractions*. A continued fraction has the form[2]

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{\cdots + \cfrac{1}{a_n}}}$$

A regular continued fraction is one in which all the coefficients except a_0 are positive, and $a_n > 1$ (except for the special case 1). Every rational has a unique representation as a regular continued fraction.

Having arrived at this elegant result, the authors proceed to reduce their formulas to the notation of Haskell for implementation, using lists of integer coefficients to represent continued fractions. In the process, the origins of the code are completely obscured by the loss of the original notation. For example, one of four cases for negation of a regular continued fraction looks like this:[3]

$$\mathit{negatecf}\ [n_0, 2] = [-n_0 - 1, 2]$$

It's up to the reader (of the paper or of the code) to decode the representation of fractions being used here and work out how this corresponds to the algebra that motivated it. However, in the proper notation, the same definition reads as simple algebraic equation which is easily understood and checked:

$$-\left(n_0 + \frac{1}{2}\right) = (-n_0 - 1) + \frac{1}{2}$$

The awkwardness of Haskell's notation is an impediment to understanding the program as an artifact, but it also obscures the program's meaning in a more subtle way. The choice of integer lists as a representation, as opposed to defining a new algrebraic data type with a similar recursive structure, was probably driven by the relative economy of the notation for lists (which is provided by the Haskell parser as a special case). As a result, accurate type information is lost, which makes the program harder to understand both in the writing and at runtime.

2.2.2 Continued Fractions in Lorax

In Lorax, one can extend the language with a new kind of value for these fractions. I did this by defining a **continuedFraction** node which defines a recursive data type. It is displayed in the obvious way, except that when the continuation is the "null" value, it is displayed in a slightly simplified form. The *expand* reduction is new—it reduces to an expression which evaluates the component expressions and then constructs a node. Therefore the node itself becomes a runtime value. Several **match** nodes provide pattern

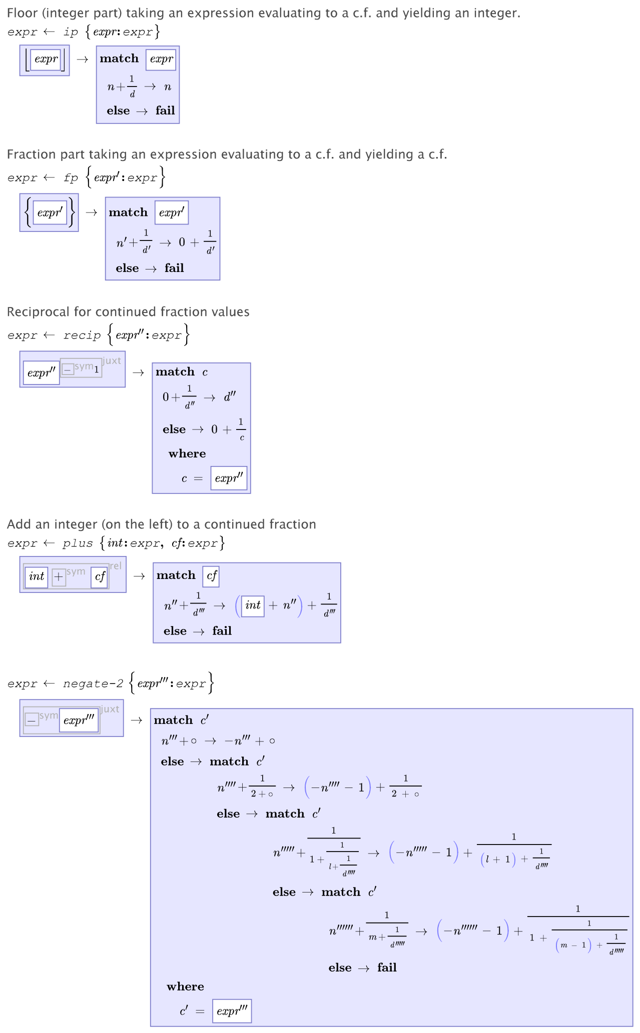
[1] In the authors' notation, $\lfloor x \rfloor$ is the floor, or whole-number part of x, and $\{x\}$ is the fractional part: $\{x\} = x - \lfloor x \rfloor, x > 0$.

[2] Incidentally, this expression is a frequently-cited exception to TEX's rules for formatting fractions—all the nested expressions are best typeset at the same size, to emphasize the recursive structure. Lorax does not provide a way to override that behavior, so continued fractions do not look quite this nice in Lorax!

[3] Actually, what's shown in the paper has been pretty-printed for publication [5]. In the actual source code, it looks even less like math:

```
negatecf [n_0, 2] = [-n_0-1, 2].
```

matching on the runtime shape of the argument, and are used to identify the cases in each operation. Figure 2 shows the declarations of the five operations.



Floor (integer part) taking an expression evaluating to a c.f. and yielding an integer.

$\mathtt{expr} \leftarrow \mathtt{ip}\ \{\mathit{expr}\colon \mathtt{expr}\}$

$$\boxed{\lfloor \boxed{\mathit{expr}} \rfloor} \rightarrow \boxed{\begin{array}{l} \textbf{match}\ \boxed{\mathit{expr}} \\ \quad n + \frac{1}{d} \rightarrow n \\ \quad \textbf{else} \rightarrow \textbf{fail} \end{array}}$$

Fraction part taking an expression evaluating to a c.f. and yielding a c.f.

$\mathtt{expr} \leftarrow \mathtt{fp}\ \{\mathit{expr}'\colon \mathtt{expr}\}$

$$\boxed{\left\{ \boxed{\mathit{expr}'} \right\}} \rightarrow \boxed{\begin{array}{l} \textbf{match}\ \boxed{\mathit{expr}'} \\ \quad n' + \frac{1}{d'} \rightarrow 0 + \frac{1}{d'} \\ \quad \textbf{else} \rightarrow \textbf{fail} \end{array}}$$

Reciprocal for continued fraction values

$\mathtt{expr} \leftarrow \mathtt{recip}\ \{\mathit{expr}''\colon \mathtt{expr}\}$

$$\boxed{\boxed{\mathit{expr}''}\ \boxed{-}^{\text{sym}}{}_1{}^{\text{juxt}}} \rightarrow \boxed{\begin{array}{l} \textbf{match}\ c \\ \quad 0 + \frac{1}{d''} \rightarrow d'' \\ \quad \textbf{else} \rightarrow 0 + \frac{1}{c} \\ \quad \textbf{where} \\ \qquad c = \boxed{\mathit{expr}''} \end{array}}$$

Add an integer (on the left) to a continued fraction

$\mathtt{expr} \leftarrow \mathtt{plus}\ \{\mathit{int}\colon \mathtt{expr},\ \mathit{cf}\colon \mathtt{expr}\}$

$$\boxed{\boxed{\mathit{int}}\ \boxed{+}^{\text{sym}}\ \boxed{\mathit{cf}}^{\text{rel}}} \rightarrow \boxed{\begin{array}{l} \textbf{match}\ \boxed{\mathit{cf}} \\ \quad n'' + \frac{1}{d'''} \rightarrow \left(\boxed{\mathit{int}} + n''\right) + \frac{1}{d'''} \\ \quad \textbf{else} \rightarrow \textbf{fail} \end{array}}$$

$\mathtt{expr} \leftarrow \mathtt{negate\text{-}2}\ \{\mathit{expr}'''\colon \mathtt{expr}\}$

$$\boxed{\boxed{-}^{\text{sym}}\ \boxed{\mathit{expr}'''}^{\text{juxt}}} \rightarrow \boxed{\begin{array}{l} \textbf{match}\ c' \\ \quad n''' + \circ \rightarrow -n''' + \circ \\ \quad \textbf{else} \rightarrow \textbf{match}\ c' \\ \qquad\qquad n'''' + \frac{1}{2+\circ} \rightarrow \left(-n'''' - 1\right) + \frac{1}{2+\circ} \\ \qquad\qquad \textbf{else} \rightarrow \textbf{match}\ c' \\ \qquad\qquad\qquad n''''' + \cfrac{1}{1 + \cfrac{1}{l + \cfrac{1}{d''''}}} \rightarrow \left(-n''''' - 1\right) + \cfrac{1}{\left(l+1\right) + \cfrac{1}{d''''}} \\ \qquad\qquad\qquad \textbf{else} \rightarrow \textbf{match}\ c' \\ \qquad\qquad\qquad\qquad n'''''' + \cfrac{1}{m + \cfrac{1}{d'''''}} \rightarrow \left(-n'''''' - 1\right) + \cfrac{1}{1 + \cfrac{1}{\left(m-1\right) + \cfrac{1}{d'''''}}} \\ \qquad\qquad\qquad\qquad \textbf{else} \rightarrow \textbf{fail} \\ \quad \textbf{where} \\ \qquad c' = \boxed{\mathit{expr}'''} \end{array}}$$

Figure 2. Grammar for operations on continued fractions as runtime values.

With the hard work of defining syntax and semantics out of the way, the actual algorithm can be expressed quite naturally. The expression which produces the next fraction in the series is wrapped in a function:

$$\textbf{fn}\ x \rightarrow \left(\left(\lfloor x \rfloor + 1\right) + -\{x\}\right)^{-1}$$

Note that in this form the expression does not exactly match what was shown earlier, because some algebraic manipulation is necessary to put it into a form that uses only the operations that have been defined. The resulting expression contains one node (operator) for each operation to be performed. For instance, the original expression hid a negation and an addition operation behind a single $-$ symbol, but my version makes the two operations explicit. Also, Lorax inserts a pair of parentheses to clarify the order of evaluation of the two addition operations, another point which is left to algebraic convention in the original. Other than that, my choice of notation resembles the original precisely.

$$\begin{array}{l} \$\ \ \mathit{rationals}\left[i\right]\ |\ i \leftarrow 0\,..\,6 \\ \quad \textbf{where} \\ \qquad \mathit{rationals} = \mathit{next}^{*}\left(1 + \circ\right) \\ \qquad \mathit{next} = \textbf{fn}\ x \rightarrow \left(\left(\lfloor x \rfloor + 1\right) + -\{x\}\right)^{-1} \end{array}$$

$$\rightarrow\ 1 + \circ,\ 0 + \frac{1}{2+\circ},\ 2 + \circ,\ 0 + \frac{1}{3+\circ},\ 1 + \frac{1}{2+\circ},\ 0 + \cfrac{1}{1 + \cfrac{1}{2+\circ}},\ 3 + \circ$$

Figure 3. Enumerating the rationals, using continued fractions.

Now generating the infinite series in continued fraction form is as simple as applying the *iterate* operator (*) to the *next* function, using 1 as the initial value:

$$\mathit{rationals} = \mathit{next}^{*}\left(1 + \circ\right)$$

The complete expression and the first 15 fractions (converted to simple ratios) appear in Figure 3.

2.2.3 Evaluation

Lorax allows a novel runtime value to be defined entirely in terms of nodes and reductions. Once a constructor and some syntax for pattern matching on the new values are defined, it's quite straightforward to implement operations on the new values, and to write programs which make use of the values and operations on them.

In the case of continued fractions, the notation is clearly superior to what can be done in a textual language, both aesthetically and in terms of ease of understanding. Furthermore, the improved notation encourages the use of a proper data type, as opposed to the awkwardness of Haskell which pushes the programmer towards using a generic data type to take advantage of a more convenient syntax.

One thing to note is the use of identical $+$ symbols for two distinct addition operations (first addition of integers, and then addition of an integer to a continued fraction). It might be preferable to use a different symbol for the new kind of addition. Because that symbol is specified in one place—the presentation reduction for that node—it's trivial to make the switch, without the need to update each use of the operator. On the other hand, this may be a case where some ambiguity in the visual representation can actually enhance readability. If that freedom leads to an incorrect program, it's easy to click on either node and the editor will indicate which $+$ is of which type. This represents a middle ground between the approach of "overloading" operators at the level of syntax, relying on the compiler or runtime to disambiguate them (as in C and many similar languages), and using a distinct operator for each type of argument (as in *[TODO: what's that language that uses .+ for adding FP numbers?]*).

However, the current limitations of Lorax's approach to reductions make the job of defining these operations more arduous than it should be. A more general mechanism for implementing pattern-matching is needed to make this attractive.

3. Lorax

In the Language Oriented Meta-Programming System,

All programs represented as ASTs using a common representation.

A couple of pre-defined languages: kernel language, expression language

Languages defined by transformation: core language, grammar language

Reductions/transformations written in the kernel language, which

4. Programs and Transformations

To build programs directly out of nodes, we need a common representation for nodes. It must be flexible enough to represent many kinds of programs and support the kind of editing operations we will want to provide. It should provide a natural way to represent the primitive values which appear in programs, and the common ways of aggregating values into larger structures. It should be able to represent arbitrary programs, and allow nodes to be freely composed without imposing any fixed constraints. Also, AST nodes will be called on to represent elements of not only source programs, but also intermediate programs, grammars, and graphical elements.

The representation we use is similar to s-expressions... *[TODO: ?]*

4.1 Nodes, Values, and References

A *program* is a tree made up of *nodes*. Every node has a *tag*, a unique *label*, and a *value*. A node's tag identifies it as one of a class of related nodes, all of which have some common meaning (for example, the tag **plus** might represent addition expressions). A node's label is an opaque identifier which gives it a distinct identity. A node's value may be: *empty*, if the node tag alone carries all the node's meaning; an atomic value, which is a *boolean*, *integer*, or *string* (a sequence of characters which are treated as an indivisible value); a *sequence* of n child nodes (indexed by integers 0 to $n-1$) or an (unordered) *map* of distinct *attribute names* and associated child nodes. A *reference* is a special kind of node which has the tag **ref**, and has as its value the label of another node.

These four kinds of nodes seem to be sufficient to represent the elements of typical programs. Of the four, map nodes are the most commonly used, and their ability to contain an arbitrary set of attributes is key to the extensibility of the system. Sequences are represented with a first-class node type primarily for the convenience of the editor, discussed later. Reference nodes allow one part of a program to refer to an entity which is declared elsewhere, rendering such references unambiguous.

Nodes are immutable values, so a program is a *persistent data structure* [9]. A modified program may be constructed by building a new tree sharing much of the structure of an existing tree, which allows an efficient system to be built out of many layers of tree transformation.

A program is *well-formed* if its nodes satisfy the following constraints:

1. No two nodes have the same label.
2. No node appears as a child of more than one parent, or under more than one index/name of a sequence/map-valued node (i.e. the nodes form a tree).
3. The value of each reference node is the label of some node in the program.

Note that any node can be considered as the root node of a sub-tree consisting of its descendant nodes. A sub-tree of a well-formed tree will be well-formed unless it contains a reference to a node which is not part of the same sub-tree. Such a reference is analogous to a free variable, and can play a similar role.

These minimal constraints allow trees to be transformed into one another using familiar functional programming techniques, and ensure that operations on trees have well-defined results. Therefore, Lorax requires that trees be well-formed at all times, and its components are designed to enforce these constraints.

4.2 Specifications

A *specification* constrains the structure of nodes in a program. A program is *valid* with respect to a certain specification if the arrangement of nodes, values, and references satisfies the specification's constraints.

In practice a program may contain violations of a specification, and the user will be interested in the nature of each violation in order to be able to fix them (by correcting either the program or the specification). Therefore a specification will typically be implemented in the form of a *checker*, a function over programs that produces a set of *errors* each consisting of a description of the problem and the location where it occurs.

Depending on the nature of the properties being checked, a specification may be defined only in terms of *local* properties of individual nodes and their direct children, or may refer to *non-local* relationships between nodes more distantly connected. In general, local properties are easier to define, easier to check, and easier to understand. In section **??**, we describe a particularly direct and convenient way of specifying these basic structural properties which is sufficient to define the abstract syntax of a programming language.

By providing a modular way of describing and checking properties of programs, specifications give much-needed structure to the open model of ASTs described in the preceding sections. They do not, however, restrict the user's ability to modify and extend her program and/or language, even if that means the program is invalid at times.

4.3 Reductions

The next step in making a useful system is providing a way to produce (multiple) *target* programs from a given *source* program. The central idea is to have a source program, consisting of nodes in some "user" programming language, which includes a node type for every important programming construct. In an extensible system, the user is able to add new types of nodes or even entire languages to a running system, even though the underlying system only understands a fixed set of node types. The way to bridge this gap is via *reduction*.

Reduction is a restricted form of *graph reduction*, inspired by the macro expansion process of Lisp.[4] In fact the term "expansion" might be more appropriate because a typical reduction replaces a more abstract node with a larger number of simpler nodes.

A *source* program is reduced to a *target* program by applying a *reduction function* to the root node of the source program. If any reduction is possible, the reduction function returns a new, replacement root node, which typically repackages the children of the original root node under some new kind of parent. As long as some reduction is performed, the reduction function is applied repeatedly to the previous result. Eventually, the root node is fully-reduced (the function fails to return a new reduced node). At that point, each child node is reduced in the same way, and a new root node is constructed with the reduced children.

4.4 Defining Specifications and Reductions

[TODO: A place to talk about defining transformations via attribute grammars and meta-language.]

4.5 Characteristics

This way of constructing program source has some implications for the way languages can be defined and the way programs can be worked with.

Node types, attributes, and specifications naturally fit with the concepts of *tree grammars*, which allows specifications to be defined in ways that are familiar to language designers (i.e. with a grammar). Because grammars so-defined will be used only for

[4] The term "Lisp" is meant to refer to any of the many variants of Lisp, including Maclisp, Common Lisp, Emacs Lisp and Scheme.

	Lorax LOC	MPS LOC
foo	100	10,000

Table 1. Comparison of estimated lines of code...

checking tree structure and not for parsing, they can be constructed in the natural way, without any need for tricks to work around parsing algorithm shortcomings (e.g. left-factoring).

Programs are *self-describing*. Each node carries an explicit declaration of its meaning, and each primitive value is manifestly of a certain type. This is in contrast to a textual language, where the same sequence of characters might represent a name, data, or a keyword, depending on the context in which they appear. This is a major advantage especially for tools that manipulate programs, because no parser is necessary to extract the structure of the program.

Labels provide robust *source locations*. The label provides each node with an identity that survives when the other parts of the program are changed, or when the program is serialized to non-volatile storage, etc. Thus labels provide a way for tools (e.g. compilers and debuggers) to refer to source locations. For example, a typical refactoring operation such as changing the name of a variable or moving some code from one place to another looks like several unrelated changes to a *diff* tool that has two versions of source text to look at, but a similar tool operating on labeled nodes could compare each *node*, even if the location, the content, and even the type of the node have changed.

Reference nodes provide a way of referring to entities in a program which can never be ambiguous, and is independent of such language-specific notions as names and scope. Reference nodes require special support from editors, which may also take advantage of the explicit reference structure to provide enhanced presentation of references.

5. Expression Language

A new language for displaying programs, mathematical expressions, and simple graphical elements.

Nodes represent visual elements at a fairly abstract level. Source program is reduced to expression language.

From $\TeX$: algorithms, symbols, inspiration.

Space is part of the language. Parentheses are inserted as needed by a generic transformation (no need for language designer or user to worry about it).

6. Evaluation

Experience with the prototype system allows some comparison to be made with existing systems.

Table 1 ...

7. Related Work

[Not the language workbenches, but other AST-based work?]

8. Conclusion

References

[1] John Cowan and Richard Tobin, editors. *XML Information Set (Second Edition)*. 2004.

[2] James Duncan Davidson. *Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Apps*, chapter The Creator of Ant Exorcizes One of His Demons. The Pragmatic Programmers, 2004.

[3] Jeremy Gibbons, David Lester, and Richard Bird. Functional pearl: Enumerating the rationals. *Journal of Functional Programming*, 16(03):281–291, 2006.

[4] Paul Graham. *On LISP: Advanced Techniques for Common LISP*. Prentice Hall, 1993.

[5] Ralf Hinze and Andres Löh. The lhs2tex system for typesetting haskell. Available at: http://www.cs.uu.nl/ andres/lhs2tex/, 2006.

[6] Allen Holub. Just say no to xml. *SD Times*, September 2006.

[7] Openoffice.org xml file format 1.0 december 2002. Technical report, Sun Microsystems.

[8] A. Quint. Scalable vector graphics. *Multimedia, IEEE*, 10:99–102, 2003.

[9] Neil Ivor Sarnak. *Persistent data structures*. PhD thesis, New York University, 1986.

[10] M. Ward. Language oriented programming. *Software–Concepts and Tools*, 15:147–161, 1994.