DFINING LANGUAGES

MOSS PRESCOTT

1. Concepts

A *language* is made up of several *sub-languages*. A *program* is made up of a tree of *nodes*. A node has a *type*, *id*, and a set of tagged fields, each of which contains a sequence zero or more nodes or primitive values. A sub-language is defined by a set of rules (productions?) which define what nodes may occupy which fields of which other nodes. A sub-language may introduce new productions for the nodes first defined in a different sub-language.

Each sub-language has different creators, programmers, and consumers. A series of *reductions* can be applied to a program to reduce it to a particular sub-language, which can be used by a particular consumer. For instance, the *editor* reduces a program to a presentation language program which can be drawn on the screen, while a meta-compiler reduces the same program to a kernel language program which can be compiled to an executable.

However, not every sub-language is required to be reducible to executable (kernel) form. For example, a symbolic algebra program might manipulate algebra expressions. There would be a sub-language for these expressions so that they could be expressed in the program, but they would never be reduced by the compiler.

A sub-language may also define additional rules which can be used to do other kinds of static checks. For example: *scoping rules*, *typing rules*, *access modifiers*, Clojure's explicit tail-recursion, etc.

Meta-programs can embed programs in other languages if a sub-language provides adapting syntax. For example, a *sql.query* node type cannot reside where a *java.expr* node is expected, but a *java.embed.sql.query* node could, and would be reduced at compile time to some ordinary Java code which constructs a Java value encapsulating the query. The embedded code would have all the editor support of a regular SQL program, and could contain unquoted Java expressions referring to variables in the enclosing scope, etc.

2. Core Language

Some common elements which may be used by any language.

`language` *core*

ref {id : `uniqueid`} — refers to another node by its unique id

quote {body : ∗} — delays evaluation of its body, which can be any node
unquote {body : ∗} — causes its body to be evaluated sooner

Date: 21 Feb. 2010.

3. "Clojure" Kernel Language

Nodes defining the base language, which is based on the primitive "special forms" of the LISP variant *Clojure* (more or less the lambda calculus). These primitives can be trivially converted to Clojure forms, so once a program is reduced to this language it can be compiled and executed.

This sub-language contains only nodes that are of interest to an (abstract) compiler, so for example there are only bindings, but no names.

`language` *clojure.kernel*

bind {} — represents a binding; no required attributes

expr — 'abstract' node type which all nodes will implement; no node ever actually has this type

expr ::= *lambda* {params : $[core.bind]^{0..n}$, body : *expr*}
lambda {params : $[x, y]$, body : b} $\overset{present}{\longrightarrow}$ $\lambda\, x, y.\, b$
(TODO: some way to define the scope of the bindings)

expr ::= *app* {expr : *expr*, args : $[expr]^{0..n}$}
$\overset{present}{\longrightarrow}$ e e_1 ...

expr ::= *let* {bind : *bind*, expr : *expr*, body : *expr*}
$\overset{present}{\longrightarrow}$ `let` x = e_1 `in` e_2

expr ::= *if* {test : *expr*, then : *expr*, else : *expr*}
$\overset{present}{\longrightarrow}$ `if` e_1 `then` e_2 `else` e_3

expr ::= *var* {ref : *core.ref*} — a reference to a binding, which must be in scope according to...

expr ::= *true* {}
expr ::= *false* {}
expr ::= *int* {value : `int`}

expr ::= *extern* {name : `identifier`} — a way to call into the language runtime

4. "Clojure" Core Language

Elements of the full *clojure.core* namespace (that is, the language as it is available to the Clojure programmer), implemented as syntax extensions reducible to the kernel language.

`language` *clojure.core*
`requires` *clojure.kernel*

expr ::= *and* {left : *expr*, right : *expr*}

and {left : l, right : r} $\overset{present}{\longrightarrow}$ l `and` r

l `and` r $\overset{compile}{\longrightarrow}$ `let` $x =$ l `in` (`if` x `then` r `else` x)

expr ::= *cons* {first : *expr*, rest : *expr*}
$\overset{present}{\longrightarrow}$ f . r

expr ::= *nil* {} — the empty list
$\overset{present}{\longrightarrow}$ []

expr ::= *list* {elements : $[expr]^{0..n}$}
$\overset{present}{\longrightarrow}$ [$e1$, . . .]

list {elements : $\emptyset$} $\overset{compile}{\longrightarrow}$ *nil* { }

[e_1] $\overset{compile}{\longrightarrow}$ e_1 . []

[e_1 , e_2] $\overset{compile}{\longrightarrow}$ $e1$. (e_2 . [])

expr ::= *for* {bind : *bind*, expr : *expr*, body : *expr*}
$\overset{present}{\longrightarrow}$ [e | x ← lst] $\overset{compile}{\longrightarrow}$ `let` $f = (\lambda\, l.($`let` $x = first(l)$ `in` e $)$. $(f\ rest(l)))$ `in` (f lst)

...many more...

5. Documentation Language

Names, comments, and other information which aid comprehension but are not used by the compiler.

`language` *clojure.core*

name {base : `chars`, primes : `int`, subscript : `chars`, locale : `localeid`}
e.g. x, *count*, x', $index_{start}$, γ

core.bind {names : $[name]^{0..n}$} — adds a new field to a node type defined elsewhere

clojure.kernel.expr {comment : *text*?} — adds a new field to a whole class of nodes, and declares it to be optional (same as $[text]^{0..1}$)

text ::= . . . — this could be a whole grammar for rich text, to support hyperlinks, embedded code snippets, “literate programming”, etc.

6. Presentation Hint Language

Extra information which helps the editor/browser display the program in a useful and aesthetic way.

clojure.kernel.int {base : `2, 8, 10, 16`} — controls how the value of an integer literal is displayed.