Building Language Towers with Ziggurat

DAVID FISHER and OLIN SHIVERS∗
Northeastern University

Abstract

Ziggurat is a meta-language system that permits programmers to develop Scheme-like macros for languages with nontrivial static semantics, such as C or Java (suitably encoded in an S-expression concrete syntax). Ziggurat permits language designers to construct "towers" of language levels with macros; each level in the tower may have its own static semantics, such as type systems or flow analyses. Crucially, the static semantics of the languages at two adjacent levels in the tower can be connected, allowing improved reasoning power at a higher level to be reflected down to the static semantics of the language level below. We demonstrate the utility of the Ziggurat framework by implementing higher-level language facilities as macros on top of an assembly language, utilizing static semantics such as termination analysis, a polymorphic type system and higher-order flow analysis.

1 Models and design

Designers work with *models* of the artifacts they propose to construct: architects work with building schematics, antenna engineers with pole plots, electrical engineers with circuit diagrams, chip designers with gate diagrams, financial engineers with multivariate CAPM portfolio models, and so forth. The power of a model lies in its ability to provide *predictive power*—it tells us about critical properties of the artifact while we can cheaply alter it "on paper," giving us the freedom to explore possibilities at design time with little commitment.

Models, themselves, are created and manipulated in some form of constraining *framework*, which defines the form of the model. The meta-task of defining these frameworks is a key enabler to doing good design: a good framework makes the design task clear and straightforward; a poor one can complicate the task or unnecessarily limit the space of possibilities.

Design frameworks have multiple important properties:

- **Expressive range**
  Does it allow us to describe the full range of artifacts we wish to construct? (We might not, for example, be able even to write down asynchronous-logic circuits using a notation designed for globally-clocked gates.)
- **Expressive constraint**
  Does it prevent us from erroneously specifying artifacts with undesireable properties? (For example, it's impossible for a program specified as a regular expression to become stuck in an infinite loop.) In other words, a model framework is as important

∗ This material is based upon work supported, in part, by the National Science Foundation's Science of Design program, under Grant No. 0757025, and by the Microsoft Corporation.

for what it does *not* allow us to say as for what it *does*. Restricting the framework is a cognitively focussing mechanism that channels us toward good designs—if, that is, our framework is well chosen.

- **Analysis**
  Can a human or an automatic tool reason about properties of the specified artifact? (How much load can the bridge carry? How fast can the airplane fly? Can we generate photo-realistic pictures of the building's exterior at dusk?)
- **Abstraction**
  Does the model allow us to suppress and delay parts of the specification deemed to be inessential to the current stage of design? (Which processor registers will hold which values? What physical transport will carry the protocol's packets? Will the amplifier handle American or European voltage standards?)

Abstraction has an interaction with analysis in the standard tradeoff between model detail and analytic power. If, for example, we wish to design an electronic circuit that will function at microwave frequency, we cannot use the standard transistor model, but must instead employ the more complex Ebers-Moll model which captures, among other things, the parasitic capacitances that become relevant at very high frequencies. The extra complexity necessary to capture the behavior of transistors operating in the microwave regime comes with cost: it's much harder to analyse the circuit and make predictions about it.

What this means is that when we design circuits to function at audio frequencies, we do *not* want to use the more "accurate" Ebers-Moll model, for the same reason we do not try to model the orbit of the planets using quantum mechanics. The simpler, less accurate model gives us better answers. Thus the ability to choose a model that abstracts away properties inessential to the intended task is an important source of model clarity and analytic power.

2 Meta-design of specialised notations

Many design tasks, software engineering among them, function primarily with *text-based* or *language-based* design frameworks. A design framework in one of these domains, then, is a language or set of languages whose syntax and semantics span the relevant design spaces. All the criteria discussed above apply to the design of these design frameworks.

Software engineering is a particularly interesting text-based design task, because the boundary between model and artifact is (usefully) vague: software is a domain where the map *is* the territory. A C program can be considered both a final product as well as a "partial specification" for the actual machine-code program that it represents. In turn, we could say the same thing of the machine-code program, given different implementations of the same instruction-set architecture (*e.g.*, a super-scalar IA32 processor from Intel, *versus* a deeply pipelined implementation from AMD, *versus* a system that does binary translation of the IA32 machine code to a Transmeta executable). This indefinite recursion is another sign of model/artifact identification.

Language designers—that is, people whose job is designing the design framework itself—exploit this identification of model and artifact. For example, a type system can be considered an auxiliary, redundant, coarse-grained behavioral model of the program. It bounds the behavior of the program by ruling out possible bad actions (run-time type

errors). By serving as a redundant specification, it acts to double-check the rest of the program: if the types and the code get out of sync, an error is revealed. So, the types in a program are simultaneously a part of the artifact—the program—as well as annotations modelling important properties of the artifact.

The basic structure of a language-based design framework is a syntax specification and an associated semantics assigning meaning to forms from the language—that is, it connects the syntactic forms (text strings or trees) to the world of artifacts they describe (bridges, computations, airplanes, *etc.*). The meaning of a design is the thing it models or describes or specifies. The semantics is usually provided in multiple phases: a "static" semantics that describes properties of the artifact that can be determined purely from the specification, and a "dynamic" semantics describing properties that may escape static determination due to the lack of an effective procedure to compute them, or dependence on the external environment. These are concepts and terms taken from the field of programming languages, but they apply to other design frameworks as well: some examples are the description of images (Abelson *et al.*, 1985), 3D objects (ISO, 2004), electronic circuits, and financial models (Peyton Jones *et al.*, 2000).

To take the example of a financial model for a stock portfolio or equity option, its "static semantics" are necessarily described in terms of statistical and probabilistic properties, since we do not know at analysis time the future of the stock market. A portfolio's "dynamic semantics" are determined by external events in the real world—that is, what the given portfolio will be worth at some point in time, or what actual value a call option will yield six months from now. The static semantics helps guide financial advisors as they construct portfolios.

The ability to choose an appropriate framework, one that hides the inessential, eliminates the undesirable, constrains and focusses one's thinking toward good designs, and allows appropriate reasoning about the artifact being designed, is a critically important part of the design process.

In this article, we will explore a tool, Ziggurat, that makes it possible for engineers to engage in meta-design—that is, to design specialised notations to aid the design process. The notation-design process happens by means of what we call "language towers."

Our goal is to provide a system that will drastically reduce the difficulty of constructing useful text-based design frameworks. While our expertise lies within the domain of software, we believe this technology could be applied to other areas as well—any design domain that can profit by having a text-based design notation that permits useful formal, static analysis of the design as expressed in this notation.

The remainder of this paper describes the Ziggurat metadesign framework. We begin (Section 3) by explaining the concept of a "language tower;" then (Section 4) outline how this topic has been addressed with Scheme macros, and discuss how this solution does not by itself generalize to other languages. Next, we give a high-level overview (Section 5) of how a language designer can use Ziggurat to construct a language tower that has associated static semantics. We then introduce (Section 6) the key model of computation behind Ziggurat, called *lazy delegation.* The next two sections discuss how Ziggurat is used to implement languages: how it is used to define an abstract syntax tree (Section 7), and how one parses terms with Ziggurat (Section 8). We then begin an extended example (Section 9) by designing an assembly language with Ziggurat. With our language defined, we demonstrate

how to perform static analyses on this language: termination analysis (Section 10) and a simple type system (Section 11). In both cases, the static semantics of the language can be extended to syntactic extensions created with Ziggurat. The next few sections build a tower up from this language, beginning with one-way continuation-passing function calls (Section 12) and an associated procedural control-flow analysis (Section 13), then introducing closures (Section 14) and direct-style functions (Section 15). We conclude by discussing possible variations on the Ziggurat design, and related and future work.

3 Language towers

The notion of a language tower arises when we are able to use a higher-level language to describe an artifact expressible in some lower-level notation by specifying a relationship between the semantics of the two languages. We typically specify the relationship between the two languages by providing a translation from the higher-level to the lower-level one. Having done so, we can shift to performing design in the high-level language, yet still realise these artifacts in the lower-level framework by means of translation.

3.1 Specialised notations as sources of understanding

The use of language towers allows us to express a design in a specialised notation. These notations are a source of information that helps us to understand the system we are designing. When we think of extending a language, we frequently think of adding new features, in the sense that C++ extends C. However, it is far more frequent to adopt a specialised notation that *removes* features. This is because of a fundamental tradeoff between the power of a programming language and our ability to analyse computations expressed in that language—restricting the expressiveness or power of the notation usually increases our ability to statically analyse the programs so expressed, and *vice versa.*

Consider, for example, the great success of regular expressions as a specialised notation. There's nothing we can do with regular expressions that we can not do with C or some other Turing-complete language. A major benefit of regular expressions lies precisely in their *lack* of power. Because they are so restricted, there are almost no questions one can ask about a regular expression or its associated automaton that cannot be answered. When we shift to Turing-complete languages, the situation is much worse: there are almost no questions we can ask that we *can* answer, in general. Program analyses for Turing-complete languages are always limited approximations that skate along the edge of intractability (if we're lucky) or uncomputability (if we're not).

When we can express a desired computation in a restricted notation such as regular expressions, it's almost always to our benefit to do so. It's easier for automatic tools to analyse the computation; it's easier for a human to write the program and get it right; it's easier for a human to look at the program and understand it. These benefits—clarity of expression and power of analysis—are related, which is one of the key reasons why domain-specific languages have become a focus of interest in recent years.

3.2 Connecting across layers of the tower

One of the key desiderata in the design of a system for constructing language towers is providing a way to relate the static semantics of adjacent layers in the tower. This enables us to project the increased analytic power we have for terms in the higher layer down to their corresponding translations in the lower layer.

Consider our regular-expression example. Suppose our regular expressions are given dynamic semantics by translating them into C code that implements the corresponding finite-state automata. Suppose, further, that we have a static analysis for general-purpose C code that determines if a given C statement is guaranteed to terminate. Since such an analysis is necessarily approximate, we clearly would be better off if we could do the analysis at the regular-expression level, where we can do a perfectly precise job (since all regular expressions represent terminating computations), and then map the answer down to the underlying C layer of the language tower.

4 Scheme macros

The most successful system for extending syntax in use today is the Lisp family's macro facility, including Scheme's "hygienic" macros. It is an everyday task for Scheme and Lisp programmers to create small, domain-specific languages to handle database queries, string searches, Unix shell scripts, VLSI design or web-based service queries, when programming in these domains.

Our goal, then, is to take Scheme's syntax-extension facility, and adapt it for use as a front end for other languages, such as Java, Standard ML, C or even assembly language. We'll commence by exploring the basic elements of Scheme macro technology, and then move to the issues that are raised when we attempt to apply it in other contexts.

4.1 Macros and S-expression languages

Scheme's concrete syntax is unusual in that it is not defined in terms of *character strings*, but in terms of *trees*, that is, list structure whose leaves are symbols and literals such as integers, strings and booleans. These trees, and their sub-trees, correspond to Scheme expressions. Thus we describe the concrete syntax of the Scheme conditional as "a four-element list whose first element is the symbol `if`, and whose other three elements are the test, consequent and alternate expressions, respectively."

This form of concrete syntax is often called *S-expressions* or *sexps*. In Scheme's syntax, symbols represent program identifiers and literals represent constants, but there are two possibilities for lists. If the first element of a list is a keyword (such as `if` or `lambda`), then the keyword specifies the syntactic form. Otherwise, the list is interpreted as a function call.

Similarly, we can define other, completely different languages using sexp-based concrete syntax. For example, we could define a regular-expression notation where

```
(: bol                  ; beginning of line
   (* (| " " "\t"))     ; zero or more space or tab chars
   ";"                  ; semicolon char
```

```
(* any)              ; zero or more chars of any kind
eol)                 ; end of line
```

represents, not Scheme code, but a regular expression that matches, in sequence, the beginning of the line, zero or more occurrences of a space or tab character, a semi-colon, and then zero or more characters up to the end of the line. We might, alternately, define an S-expression grammar for Unix process notation, so that

```
(| (gunzip -c notes.txt.gz)
   (spell)
   (lpr -Pgaugin))
```

creates a three-process pipeline that uncompresses a file, spell checks it, and sends the spelling errors to the printer.

The macro facility in Scheme and Lisp permit the programmer to define new keyword-marked syntactic forms. The new keyword is tagged with code that is executed by the compiler: when the compiler encounters an expression of the form

(*keyword subform*$_1$ ...)

it passes the entire expression (as a tree) to the keyword's associated code, which is responsible for translating the entire form into some other expression. This is the macro-expansion step; it is repeated as necessary until the entire program is nothing but core Lisp or Scheme syntax.

We can use this facility simply to provide a new form in the language. For example, we might define an "arithmetic if" form that branches one of three ways depending on the sign of its first expression by tagging the keyword `arith-if` with code to translate

(`arith-if` *exp neg-arm zero-arm pos-arm*)

into

```
(let ((tmp exp))
  (if (< tmp 0) neg-arm
      (if (= tmp 0) zero-arm
          pos-arm)))
```

More ambitiously, we can use Scheme's macro facility to embed an entire *language* within Scheme, by arranging for the macro to be a compiler that translates terms in the embedded language to an equivalent term in Scheme. For example, we might have a macro `rx` that translates regular expressions written in an S-expression grammar (as in our example above) to Scheme code implementing a string-matcher for that regular expression:

```
(let ((matcher (rx regexp-term)))
  ... (matcher str) ...)
```

This is how we can embed arbitrary domain-specific languages within Scheme or Lisp—assuming that they are represented using an S-expression (or tree-based) concrete grammar.

4.2 Scheme macros, hygiene and lexical scope

One of the key features of the Scheme macro facility (which is not found in the older Lisp macro systems) is that Scheme macros are 'hygienic," in that they respect Scheme's lexical scope. The notion of lexical scope has a subtle interaction with macros; in particular, it has two main implications.

First, suppose we declare a macro keyword in a Scheme program, with, for example, the Scheme form

```
(let-syntax ((keyword  macro-definition))
  body)
```

In the body of the `let-syntax` form, occurrences of the identifier *keyword* refer to the defined macro; the syntax binding shadows any meaning that *keyword* might have outside the `let-syntax` form. Similarly, forms occurring within *body* that themselves bind the identifier *keyword* will lexically shadow the macro definition. Note that *keyword* can be *any* identifier, including one that is bound elsewhere as a regular variable, or even one that, at the top level of the program, is used to mark core language forms, such as `if` or `lambda`: in Scheme, there are no "reserved" words at all; there are only identifiers with lexical scope.

Second, note that our macro has two significant lexical contexts: the one pertaining at its point of *definition*, and the one pertaining at its point of *use*. In our example above, if our *keyword* macro is used somewhere inside the *body* expression, the lexical context at that point may be quite different from the context where the *keyword*/*macro-definition* binding was made.

When the macro executes, it expands into Scheme code that itself contains identifiers. Should these identifiers be resolved, in turn, using their meaning at the macro's point of definition, or using their meaning at the macro's point of use? In fact, we need both. Consider our `arith-if` macro defined above. Referring back to its expansion, we can see that the macro produces a sexp tree containing the identifiers `let`, `tmp`, `if`, `<`, and `=`, plus all the identifiers occurring in the sub-trees *exp*, *neg-arm*, *zero-arm* and *pos-arm*. The macro writer intended for all the identifiers in the first part of this list to mean what they mean at the macro's point of definition, which almost certainly includes these meanings:

- `let` should mean the keyword for Scheme's basic variable-binding form;
- `if` should mean the keyword for Scheme's core conditional form; while
- `<` and `=` should mean the top-level variables bound to Scheme's numeric-comparison functions.

It would break the macro if bindings appearing in *body* were accidentally to shadow these bindings. That is, it would be a bad thing if a binding of `tmp`, or `<`, or `if` occurring in *body* caused `arith-if` to break, *e.g.*:

```
(let-syntax ((arith-if macro-definition))
  ...
  (let ((< gregorian-date-less-than))
    ...
    (arith-if (* x y)
      (- x 3)
      0
      (+ x 3))
    ...))
```

Further, we would not want to require `arith-if` clients to avoid binding any of these identifiers; it should be invisible to the `arith-if` client how the form is implemented.

On the other hand, lexical scope also means that we want all identifier references appearing in the *exp*, *neg-arm*, *zero-arm* and *pos-arm* sub-expressions to be resolved in the context where they actually appear in the original source code—that is, at the point of the macro's use. Suppose, for example, that the *body* expression of our `let-syntax` form itself contained a binding of the variable `tmp`, *e.g.*

```
(let-syntax ((arith-if macro-definition))
  ...
  (let ((tmp (- x 3)))
    ...
    (arith-if (* x y)
      (- x tmp)
      0
      (+ tmp 3))
    ...))
```

When the `arith-if` form expands, it will introduce its own binding of `tmp` (marked with an underline):

```
(let-syntax ((arith-if macro-definition))
  ...
  (let ((tmp (- x 3)))
    ...
    (let ((tmp (* x y)))
      (if (< tmp 0) (- x tmp)
          (if (= tmp 0) 0
              (+ tmp 3))))
    ...))
```

It would break the program if bindings introduced by `arith-if` were accidentally to shadow the client's ability to bind and reference its own `tmp`. Again, we would not want to require `arith-if` clients to avoid binding `tmp`; it should be invisible to the `arith-if` client how the form is implemented.

Thus, we want (1) identifiers introduced by the macro (*e.g.*, `let` and `<`, in our `arith-if` macro) to have the meaning that they have at the point where the macro is defined, and

(2) identifiers appearing within the macro use's subforms (*e.g.*, the `*`, `x` and `y` references appearing within the *exp* sub-tree) to have the meaning that they have at the point where the macro is used. "Hygiene" is the means by which these bindings and references are kept sorted out.[1] We will return to the mechanisms that provide hygiene, but the key point to observe here is that what the Scheme community calls "macro hygiene" is nothing more than correctly providing lexical scope for macros. This allows programmers to reason about their macro definitions with the solid assurances that come with lexical scope: it is always possible to resolve an identifier reference in a Scheme program simply by looking at the point in the code where the reference occurs.

4.3 Parsing in the presence of hygienic macros

The classical view of compiler construction holds that we first lex and parse our program into a syntax tree, and then—after parsing is completed—we implement our language's static semantics with algorithms that process the tree, resolving variable references, checking types, and so forth.

Scheme's lexically-scoped (or hygienic) macros, however, require that we abandon this simple picture: parsing, in Scheme, must be intertwined with static analysis. This is necessary because Scheme abandons the notion of the fixed "reserved keyword" in favor of lexically-scoped keywords. Consider, then, what is required when the parser attempts to parse the form `(if x 4 5)`. It must resolve the identifier `if` to determine where in the program it is defined. Perhaps it is the top-level `if`—that is, it is the keyword for Scheme's basic conditional form. But there are no reserved tokens in Scheme, so perhaps it is instead a reference to a *variable* bound by some intermediate `let` or λ form, *e.g.*,

```
(λ (x if z) ... (if x 4 5) ...)
```

in which case our `(if x 4 5)` form is *not* a conditional, but is instead a function call. Or perhaps the `if` reference is a reference to a macro the programmer bound with an intermediate `let-syntax` form:

```
(let-syntax ((if macro-definition))
  ... (if x 4 5) ...)
```

We must resolve the `if` reference and determine where it was defined in the current scope *before* we can parse the `(if x 4 5)` form in which it appears. But this is a static analysis problem: lexical scope (which is the rule by which we are resolving identifier references) is part of a language's static semantics.

Thus we have a circularity not occurring in more classical compilers: we need a parse tree in order to perform static analysis, but we need to do static analysis in order to parse. This is why Scheme's provision of macros with lexical scope requires parsing to be interleaved with analysis.

[1] It's also useful to have a controlled ability to violate hygiene. For example, we might wish to define a macro that causes its body to be evaluated in an augmented scope that binds the identifier `abort` to a function; calling this function during evaluation of the body triggers a non-local exceptional transfer from the entire form. Thus the macro must introduce a binding for `abort` which is visible to its subform, hiding any outer binding of `abort`. Scheme macro systems provide mechanisms by which this can be managed.

4.4 Hygiene mechanisms

The simplified core of what a macro does is that it (1) substitutes its arguments into some template, and then (2) substitutes the filled-in template at the point of the macro use. The identifiers in the *template* appear in the program at one lexical context (where the macro is defined), while the identifiers in the *arguments* appear in the program at a different, inner context (where the macro is used). The essential requirement of hygiene is that we perform these substitutions in a manner that preserves lexical scope. This is a problem that was solved at the birth of the λ-calculus, in the form of specifying precisely how β-reduction performs substitution (Church, 1941; Barendregt, 1984; Baader & Nipkow, 1998). Hygiene mechanisms in macro systems are simply mechanisms that employ the substitution machinery from the λ-calculus, instead of using naïve textual substitution—for the same reason that β-reduction does not employ naïve textual substitution. The core of the λ-calculus's substitution mechanism lies in its ability to α-rename identifiers that might capture or be captured to *fresh names* that cannot possibly interfere with other, distinct bindings of the original name. Thus, all Scheme macro systems that provide hygiene do so by means of some kind of renaming capability.

One such mechanism is Rees's explicit-renaming system (Clinger, 1991; Clinger & Rees, 1991). To explain explicit renaming, suppose that we have a compiler recursively walking the S-expression source-code tree, parsing it and expanding macros. As the compiler recurs through the source tree, it keeps track of a symbol table, which is used to resolve identifiers. Looking up an identifier in a symbol table produces the static definition of the identifier in a given lexical context; these definitions include meanings such as "the core-Scheme `if` conditional form," "the second parameter of λ term t_{37}," "the third variable bound by the `letrec` term t_{82}" or "a macro with such-and-such definition," where t_{37} and t_{82} are nodes in the syntax tree. When the compiler encounters identifier-binding forms, such as `let`, λ, `letrec` or `let-syntax` forms, it adds the new bindings to the symbol table, and recurs into the body of the binding form with the augmented table.

In explicit renaming, the right-hand side of a macro definition is written as Scheme code that takes three arguments, *e.g.*

```
(let-syntax ((arith-if (λ (exp rename compare) ...)))
  ...)
```

When the compiler encounters a use of the macro, `(arith-if ...)`, it invokes the associated procedure, passing it the entire macro form as the first argument `exp`. The `rename` argument is bound to a special function which provides the renaming capability. It additionally captures, as we'll see, the lexical context where the macro was defined, for use as the macro expander executes at the inner, macro-use context.

This renamer function has two key properties. First, when applied to an identifier, it guarantees to return a fresh identifier that occurs nowhere in the program. Suppose that the macro's expander procedure applies the renamer function to the symbol `=`, *e.g.*, `(renamer '=)`. It is guaranteed to get an identifier *id* that is completely fresh. Thus, if our macro constructs a source tree containing occurrences of *id*, it can be sure that these occurrences can not be accidentally captured by other code in the program; references to such an identifier can only be references inserted by the macro itself.

But this leaves the question: when the compiler later attempts to parse the result of our macro expression, which contains a reference to *id*, to what will *id* refer, when it is looked-up in the symbol table? This is the second key property of the renamer function. The renamer function has access to the symbol table that describes the lexical context at the macro's point of definition. When passed the symbol `=`, it looks up the meaning m of `=` in this symbol table, then inserts an $id \mapsto m$ binding into the outermost, top-level contour of the symbol table. So if, in the future, the compiler encounters a free reference to the identifier *id* at some point in the code, it will resolve to the same thing at *that* point that `=` resolves to at the macro's *definition* point. So the macro writer can confidently insert *id* into constructed code and know that no matter what its lexical context might be, *id* will serve as a suitable "synonym" for the top-level `=` variable—this will be true even in some context that locally binds `=` to some other meaning. Thus, the renamer function gives the macro access to the scoping context that pertains at its point of definition; it permits the macro to rename identifiers from this context away to fresh names that have the same meaning, where they cannot subsequently be accidentally captured by the vagaries of the client code where the macro is used.

Note, however, that if the code produced by the macro expansion itself chooses to bind the fresh identifier *id*, then this local definition *will* shadow the top-level binding inserted in the symbol table by the renamer function. Here is our `arith-if` macro, then, written using explicit renaming:

```
;;; Call this let-syntax form's lexical context lc_ls.
(let-syntax ((arith-if (λ (e r c)
                         (let ((exp      (cadr e))   ; Disassemble
                               (neg-arm  (caddr e))  ;  source
                               (zero-arm (cadddr e)) ;  tree
                               (pos-arm  (car (cddddr e))))

                               (%let (r 'let)) ; Make fresh synonyms
                               (%if  (r 'if))  ;  for these ids --
                               (%<   (r '<))   ;  with their
                               (%=   (r '=))   ;  lc_ls meaning.

                               (%tmp (r 'tmp)) ; Fresh var.

                           `(,%let ((,%tmp ,exp))
                              (,%if (,%< ,%tmp 0) ,neg-arm
                                    (,%if (,%= ,%tmp 0) ,zero-arm
                                          ,pos-arm)))))))
  ...)
```

Thus, the macro's renamer function `r` is used to get synonyms for the identifiers we'd like to reference from the macro's point of definition (`let`, `if`, `<` and `=`), and is also used simply to generate a unique identifier to serve the role of the `tmp` variable. (Note that we have not discussed the purpose of the transformer function's third argument, the "compare" procedure. It only comes into play with more sophisticated transforms.)

Another approach involves the use of annotating identifiers in the source tree with marks

(sometimes called "paint") to α-rename entire sub-terms during macro processing (Dybvig *et al.*, 1992); the `syntax-case` system is the chief example of such a macro system (Dybvig, 1992). As with the explicit-renaming mechanism, the point of the α-renaming is to make the *textual* substitution performed by the macro expansion conform to the kind of *reference-preserving* substitution we find in the λ-calculus. The basic idea is to begin macro expansion by renaming the source sub-trees comprising the macro's "arguments" or sub-terms (*e.g.*, the *exp*, *neg-arm*, *zero-arm* and *pos-arm* sub-trees in our `arith-if` example). That is, every identifier occurring in the macro-use's source tree is marked with a new, fresh mark, effectively α-renaming them. This mark-annotated source tree is then given to the macro's associated source-to-source transform function. If these marked sub-terms are incorporated into the macro's result, their unique marks distinguish them from identifiers introduced by the macro itself. After the macro has assembled its complete result term, the macro system walks the term, where it *toggles* the fresh mark added previously. That is, we (1) remove the marks from these incorporated sub-trees, and (2) mark the identifiers that the macro itself introduced. Part (1) causes these identifiers to return to their original form, so that they are restored to whatever meaning they had in the lexical context where the macro appeared. Part (2), however, ensures that identifiers introduced by the macro are renamed away to unique names, ensuring they will not interfere with part (1)'s local identifiers.

4.5 Problems adapting Scheme macro technology to other languages

Macro hygiene makes it straightforward to correctly construct general syntax extensions in Scheme. The next logical step is to take the macro mechanism, "peel" it from the Scheme language, and apply it other languages. However, the system is narrowly adapted to the specifics of Scheme, in ways that make it difficult to apply the technology to other languages.

Focus on expressions The Scheme language has an extremely spare grammar in the following sense: its S-expressions represent very little besides expressions. Thus, in Scheme we can restrict macros to the syntactic context of an expression, and that will cover nearly all uses we might wish to make of them. (Some examples of the few syntactic forms that would *not* be covered are the variable/initial-expression bindings in a `let` or `let*` form, the parameter list in a `lambda` form, and the arms of a `cond` conditional form. We cannot, in Scheme, write macros for these syntactic elements.)

By way of contrast, consider what would be needed if we added macros to a version of the C language with a sexp-based concrete syntax. Unlike Scheme, the expression is not the overwhelmingly dominant syntactic form in C. We would wish to allow macros to appear in many syntactic contexts: expressions, statements, declarations, types, initialisers, and so forth.

Little static semantics Even more problematic is that Scheme's static semantics is as spare as its syntax. The only real static semantics provided by the language is lexical scope: the ability to resolve an identifier reference to its point of definition. As we've seen, this is reflected in Scheme's macro system: hygiene is precisely the mechanism one needs to

control identifier scoping during macro expansion. It's not a problem that there is no other mechanism in the Scheme macro system to reason about static semantics, because Scheme does not have any other static semantics about which to reason.

This is a serious problem for languages with more static semantics. Suppose, for example, that we implemented a Scheme-like macro system for a variant of Standard ML written with an S-expression concrete syntax. Loops in SML are typically written with tail-recursive function calls, but a programmer might wish to implement an SML version of Scheme's `do` loop, so that expressions of the form

```
(do ((var1 initial1 update1) ...)
    (end-test final-value)
  body)
```

would expand into

```
(letrec ((loop (λ (var1 ...)
                  (if end-test final-value
                      (begin body
                             (loop update1 ...))))))
  (loop initial1 ...))
```

How should we type-check programs written in our macro-augmented SML dialect? One way would be simply to expand all macros, and then type-check the residual core-SML program. This would work, in the sense that it would guarantee the type safety of the program. But in a more practical sense, it works only when the programmer makes no mistakes: type errors are reported to the programmer not in terms of the original source code, but in terms of its macro-expanded residue, which might be an incomprehensible mess of low-level code bearing little obvious relationship to the original source.

The compiler needs the ability to reason about the program as it is written by the programmer. To return to our example above, the compiler needs a type rule for `do` forms, just as it has type rules for `if` forms and λ-expressions; then it can type-check the program and report type errors in terms of the original code the programmer wrote. In other words, we need the ability to associate static semantics with our extensions, something Scheme macro technology does not provide. With Scheme macros, the static and dynamic semantics of a new form are given implicitly by means of translation—*i.e.*, specification by compiler. The translation mechanism is opaque, being defined procedurally either in Scheme itself, or by means of the Turing-complete pattern-matching language used for Scheme's "high-level" macros. As the macros get larger and more complex, *e.g.*, providing object systems, database-query languages or parser generators, there is no hope at all that an automatic system can extract much useful static information from their specification in standard Scheme technology.

4.6 Adding static semantics to macros

Ziggurat extends the basic technology of Scheme's hygienic macros, allowing macros to be tagged not only with code that provides its dynamic semantics by means of translation, but also with code that provides its static semantics. Our approach is twofold:

- Ziggurat employs a specialised object system, called *lazy delegation.* Syntax nodes are represented as objects, and analyses are done as methods on these objects. If a particular analysis is undefined for a class of syntax node, then the analysis is delegated to a macro-rewritten form of the syntax object.
- Analyses are structured monadically. This solves a problem in building complex semantic analyses incrementally: it is difficult to do a global analysis that requires information to flow through the syntax tree in ways that do not correspond to the sequencing associated with a recursive walk through the tree, *i.e.*, not obeying a simple propagation pattern, such as bottom-up. In Ziggurat, this is solved by arranging for the syntax nodes locally to provide higher-order analysis constructors. That is, we structure the analysis as a set of syntax-node methods, each one of which takes in a partial analysis, and returns the analysis augmented with the part for that node.
  We also provide two standard libraries to help the meta-designer define static semantics for languages constructed with Ziggurat: Kanren, a Prolog-like logic language (Friedman *et al.*, 2005; Byrd & Friedman, 2006), and Tsuriai, a system we developed for computing fixed points of recursively defined, monotone functions on lattices. In the context of language semantics, Kanren is useful for reasoning in typeful ways; Tsuriai is a framework that makes it straightforward to work in the flow-analysis paradigm. Neither of these subsystems is primitive; they are simply libraries defined in Ziggurat for the convenience of the Ziggurat programmer. There's nothing to prevent an ambitious Ziggurat programmer from implementing other libraries of a similar assistive nature. For example, it might be useful to provide programmers with a library to assist programmers in constructing abstract interpretations (Cousot & Cousot, 1977), or an implementation of the extensible HM(X) type system (Pottier & Remy, 2005), which permits language designers to implement Hindley-Milner style type reconstruction, parameterised over different base types.

Using this semantic extension capability, it is possible to layer one language on top of another, thus building up a "tower" of languages. In this way, Ziggurat is a tool for the incremental development of programming languages.

5 Designing a language with Ziggurat

Languages in Ziggurat are meant to be designed in stages, thus providing an opportunity for several actors to contribute (or for one person to play several roles). Figure 1 shows a sample workflow for building a language in two stages, but this is by no means the only way to go about it. Languages can be layered as deeply as is needed, and Ziggurat is designed to allow for complex collaboration between layers.

Consider the three characters of Figure 1. Let us assume they are building a software system in assembly language. The first character is a programmer, who will be the eventual consumer of our language. Another is a language designer, who will focus on producing a low-level assembly language. The programmer will probably not want to program in the raw assembly language, so we introduce a third character: the macro programmer. The macro programmer writes new control primitives and data operations in order to make the application programmer's life easier.

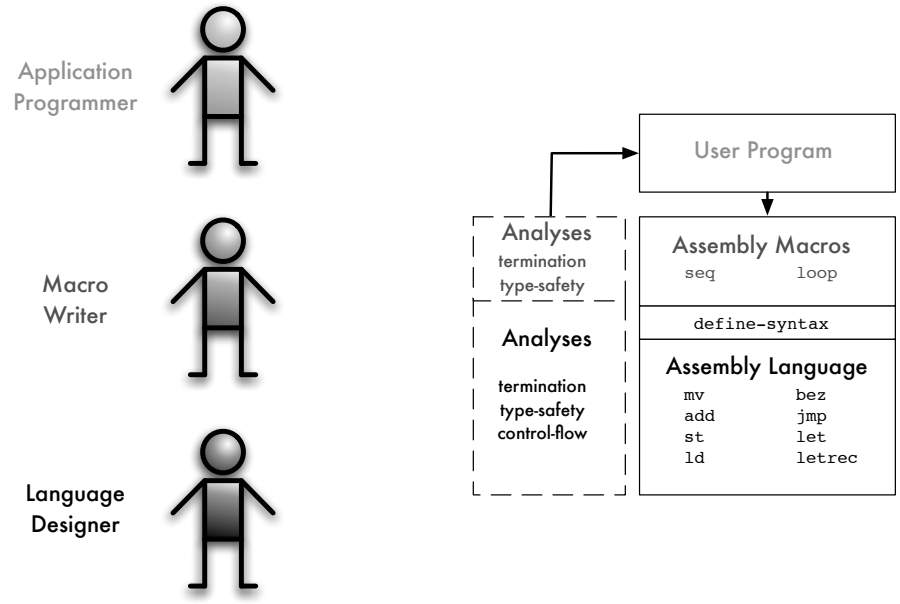

Fig. 1. Workflow to design a language with Ziggurat

We will introduce an example assembly language later in the paper. (We've purposefully chosen assembly to have an example language that is as far from Scheme as we can arrange.) Since this will not be the final language that the application programmer will be using, the language designer is free to make the language very simple. A language with a small semantics has several advantages: it can be easy to implement, reason about, and analyse. In this paper, the lowest-level language we use is an assembly language, with register and control manipulation instructions similar to a machine language. The macro writer adds additional forms, such as nested blocks and loops, and `cons`, `car` and `cdr` forms for constructing and manipulating simple tuples.

This is enough for the application programmer to start programming. However, we are not yet done; the language designer will also want to specify static semantics for his assembly language. Thus, he specifies a type system for the language, and implements algorithms for termination analysis, type reconstruction, and control-flow analysis.

At this point, though, the macro writer has information that the language designer does not have. The programmer may use structured control operations, which can provide more precise information about control flow; similarly, data operations can provide more precise type information. So, the macro writer extends the analyses of the language designer to better account for the new forms.

In order to see how this all works, we show the system from the ground up. We start with the object system of Ziggurat, which uses a mechanism for method resolution we call *lazy delegation*.

$i, c, f, m \in$ Identifier ::= `x, a, foo, ...`
$d \in$ Definition ::= ...
| `(define-class` c `(`f `...)` $[e_{\text{deleg}}]$`)`
| `(define-generic (`m i `...)` $[e_{\text{default}}]$`)`
| `(define-method` c m e`)`
$e \in$ Expression ::= ...
| `(object` c e `...)`
| `(pass)`

Fig. 2. Ziggurat extends Scheme with lazy-delegation objects

6 Lazy delegation

Ziggurat is used by the language designer to define the low-level language, and by the macro designer to extend the semantics of that language. The Ziggurat language is based on Scheme, to which we add an object model, called *lazy delegation.* Lazy delegation is similar to other delegation-based object systems, such as Self (Ungar & Smith, 1987), with a twist: the parent of an object, instead of being provided at object-creation time, is calculated only when it is needed. The additions that Ziggurat makes to the standard Scheme grammar are presented in Figure 2.

In Ziggurat, each object has a unique class. A class defines (among other things) the fields that an object has. If we wanted to define a class to represent floating-point real numbers, we would use the code

```
(define-class real (mantissa exponent))
(define real-seven (object real 7 0))
```

The first line declares that objects of class `real` have two fields: `mantissa` and `exponent`. The second line declares a variable `real-seven` to be an object of class `real` with mantissa `7` and exponent `0`.

Generic functions are introduced with the form

`(define-generic (`m i `...)` $[e_{\text{default}}]$`)`

which defines a generic function named m with the $i \ldots$ providing the argument parameters. The optional body e_{default}, if it is present, defines the default behavior of the function. Generic functions must have at least one argument, which is the "this" object of the function invocation. Generic functions are called as if they were ordinary Scheme functions, and can be passed around and used as such. Thus, the following code defines a generic function of one argument, `num`, whose default behavior is to return the string `"<object>"`.

```
(define-generic (object-number->string num)
  "<object>")
```

In order to give the generic function more interesting behavior, we must override it with class-specific method definitions:

```
(define-method real object-number->string
  (λ (this) (string-append (number->string mantissa) "e"
                           (number->string exponent))))
```

This code defines the behavior of the `object-number->string` generic function if its first argument is an object of class `real`. Since we know that the first argument of this method is an object of class `real`, we know that it has fields `mantissa` and `exponent`. These fields are available in the method body, as if they were ordinary Scheme variables. One consequence of this is that classes cannot be first-class values, since we must be able to tell statically what is a field of the object, and what is a regular variable. Objects and generic functions are first-class values, though, and can be passed around and manipulated just like any other Scheme value.

Objects have a special method, the delegate-instantiation method. Unlike other methods, delegate-instantiation methods have no arguments. A delegate-instantiation method must return an object or `#f`. This object is the *delegate* of the current object, and is used in case a generic function is applied to the object and it has no corresponding defined method. If the delegate-instantiation method returns `#f`, the object has no delegate, and any method the object attempts to delegate will fall through to the generic function's default behavior. Delegates are thus created on demand and then cached; after an object has invoked its delegate-instantiation method, future method lookups will automatically be passed to the delegate object when needed.

The delegate-instantiation method is provided as part of the `define-class` form. For example, to define an `int` class that delegates to a `real` object:

```
(define-class int (value)
  (λ () (if (= value 0)
            (object real 0 0)
            (let* ((exponent (inexact->exact
                              (floor (/ (log (abs value))
                                        (log 10)))))
                   (mantissa (exact->inexact
                              (/ value (expt 10 exponent)))))
              (object real mantissa exponent)))))
```

What if we called `(object-number->string (object int 4007))`? Since the `int` class does not have its own method for `object-number->string`, it delegates the generic function to the `real` object produced by its lazy-delegation method, which in turn produces `"4.007e3"`. If we wanted integers to have a specialised `object-number->string` method, we would simply define one, *e.g.*

```
(define-method int object-number->string
  (λ (this) (number->string value)))
```

Now `(object-number->string (object 4007))` will return `"4007"`.

Ziggurat also allows the programmer to delegate explicitly through the function `pass`. The return value of the function `pass` during a function call is the same as calling the generic function on the delegate object. For example, if we wanted an `int` object to use scientific notation when its value is less than zero, we would define the method `object-number->string` on objects of class `int` to be

$v \in$ Var ::= `a` | `b` | `c` ...
$c \in$ Const ::= ... `-1` | `0` | `1` ...
$e \in$ Expr ::= v
| c
| `(let ((`v e`) ...)` e`)`
| `(+` e e`)`
| `(*` e e`)`
| `(/` e e`)`
| `(-` e e`)`

Fig. 3. An arithmetic-expression language

```
(define-class arith-var (name))
(define-class arith-const (value))
(define-class arith-let (vars exprs expr))
(define-class arith-add (left right))
(define-class arith-mul (left right))
(define-class arith-div (left right))
(define-class arith-sub (left right))
```

Fig. 4. Implementation of the arithmetic-expression language

```
(define-method int object-number->string
  (λ (this) (if (< value 0)
                (pass)
                (number->string value))))
```

The use of `pass` here specifies that the delegate will be invoked if and only if the integer value is less than zero.

7 Representing abstract syntax trees

Building abstract syntax trees is a simple matter in Ziggurat. Consider a language for expressing simple arithmetic calculations, presented in Figure 3. This language has one basic syntax class: the expression. An expression is either a variable, a constant, a primitive operation such as multiplication or addition, or a `let` form to bind variables.

In order to implement this in Ziggurat, each syntax node is represented by an object. The classes of the syntax nodes correspond to the productions of the grammar, and the fields of the syntax node correspond to the sub-terms of the production. Implementing this involves directly transcribing the grammar into code, as seen in Figure 4.

In the `let` form, we have two fields to represent bindings: a list of variables, and a list of expressions. The two lists are linked by their order; *e.g.*, the third item in the list of variables is the variable bound to the value of the third item in the list of expressions. This is in opposition to the concrete syntax of a let statement, which takes the form of a list of pairs of labels and expressions.

This language is fairly basic, and there are any number of extensions we might want to add. For example, we might want to add a `sqr` form, that squares the value of an expression. With our AST represented by lazy-delegation objects, this is easy to do:

```
(define-class arith-sqr (expr)
  (λ ()
    (let ((temp-var-name (gensym))) ; create fresh var
      (object arith-let `(,(object arith-var temp-var-name))
                        `(,expr)
              (object arith-mul
                      (object arith-var temp-var-name)
                      (object arith-var temp-var-name))))))
```

Suppose that we have a generic function `(execute` *expr ctx*`)`, defined on the basic language nodes, that calculates the numerical value of the arithmetic expression *expr* in the variable-binding context *ctx*. Evaluating

```
(execute (object arith-mul (object arith-const 3)
                           (object arith-const 7))
         empty-arith-context)
```

would return the value `21`. Now, what about running `execute` on a syntax node of class `arith-sqr`? Consider what would happen if we attempted to evaluate

```
(execute (object arith-sqr (object arith-const 5))
         empty-arith-context)
```

Since `arith-sqr` does not define an `execute` method, it gets delegated to the expanded syntax tree:

```
(object arith-let '(,(object arith-var 'g300))
                  '(,(object arith-const 5))
        (object arith-mul (object arith-var 'g300)
                          (object arith-var 'g300)))
```

When `execute` is delegated to this AST, the result is the expected `25`. Although we did not explicitly write an execute method for `arith-sqr`, that piece of semantics was handled though rewriting.

8 Parsing

Up to this point, we've looked at programs built as Ziggurat objects with Ziggurat code. But programmers do not write programs this way; they write concrete syntax that must be parsed into corresponding Ziggurat objects.

In our S-expression setting, we have a reader that takes in a stream of characters, and produces a tree of raw data that corresponds to the nested list structure of the unparsed code. For example, the reader takes the string "`(* (+ 1 2) 3)`," and produces the data structure constructed by

```
(cons '* (cons (cons '+ (cons 1 (cons 2 '())))
               (cons 3 '())))
```

The parser is a function that takes the output of the reader and produces a tree of Ziggurat objects based on the abstract syntax of the language being implemented. For our arithmetic language, we will define a generic Ziggurat function, `parse-arith-expr`, that will take, for example, the list `(+ x 3)`, and construct the syntax tree:

```
(object arith-plus
        (object arith-var 'x)
        (object arith-const 3))
```

We could implement `parse-arith-expr` as a function of one argument, such that evaluating `(parse-arith-expr '(+ x 3))` would construct the term's syntax tree as above. However, this means of parsing makes it difficult to add a new keyword, and thus we introduce syntactic environments.

8.1 Syntactic Environments

A syntactic environment is a lazy-delegation object that defines one or more parse methods. For our arithmetic language, we will define a syntactic-environment class, `arith-env`. This class defines a method `parse-arith-expr`. The parse method is invoked with two arguments, `(parse-arith-expr` *e s*`)`, where *e* is a syntactic environment, and *s* is the form to be parsed. The class `arith-env` can naïvely be defined as

```
(define-class arith-env ()
  (λ () #f))
```

We will expand this definition shortly.

When the generic function `parse-arith-expr` is invoked on a parse environment of class `arith-env`, the environment's method extracts a keyword from the form it is parsing, and then uses a specialised parse function based on that keyword. We could define `parse-arith-expr` as

```
(define-generic (parse-arith-expr env form)
  (error "Could not parse arithmetic expression" form))
```

```
(define-method arith-env parse-arith-expr
  (λ (env form)
    (if (pair? form)
        (if (symbol? (car form))
            (case (car form)
              (('+) (parse-arith-add env form))
              (('-) (parse-arith-sub env form))
              (('*) (parse-arith-mul env form))
              (('/) (parse-arith-div env form))
              (('let) (parse-arith-let env form))
              (else (pass)))
            (pass))
        (cond ((symbol? form) (parse-arith-var env form))
              ((integer? form) (parse-arith-const env form))
              (else (pass))))))
```

While this would certainly parse the language we have described, it is too inflexible for our purposes: in order to add a new keyword, we would have to alter the definition of the generic function. So, in order to fix this, we alter the definition of `arith-env` to include a field `keywords` that contains an association list mapping keywords (symbols) to parse functions:

```
(define-class arith-env (keywords)
  (λ () #f))

(define-method arith-env parse-arith-expr
  (λ (env form)
    (if (pair? form)
        (if (symbol? (car form))
            (let ((parse-fun (assq (car form) keywords)))
              (if parse-fun
                  ((cadr parse-fun) env form)
                  (pass)))
            (pass))
        (cond ((symbol? form) (parse-arith-var env form))
              ((integer? form) (parse-arith-const env form))
              (else (pass))))))
```

(The Scheme `assq` function will search the list of keyword/parser pairs, and return the first match it finds.) For this new parse method, in order to define a new keyword, all we need to do is alter the `keywords` field of an object of class `arith-env`. We would define a top-level syntactic environment that contains the basic keywords of the arith language:

```
(define top-level-arith-env
  (object arith-env `((+ ,parse-arith-add) ...)))
```

However, this is still too inflexible for our purposes. What if we alter the language to permit local syntax declarations, similar to `let-syntax` in Scheme?

For this, we permit layering of syntactic environments. We alter the class `arith-env` to define a field `super`. The field `super` contains the syntactic environment to which the current `arith-env` will delegate if it does not recognise the current keyword.

```
(define-class arith-env (keywords super)
  (λ () super))
```

Now, to define a new keyword locally, we can create a new syntactic environment that contains the keyword in its keyword table. For example, if we wanted to define a keyword `sqr`, we would write

```
(define sqr-arith-env
  (object arith-env `((sqr ,parse-arith-sqr))
                    top-level-arith-env))
```

Upon defining this, we can invoke `parse-arith-expr` with `sqr-arith-env` if we wish to recognise the `sqr` form, or `top-level-arith-env` if we do not.

In Ziggurat, we provide a form `define-keyword-syntax-class` that automates the process of writing such a parse function. Invoking

```
(define-keyword-syntax-class syntax-class-name
  parse-function-name environment-name
  no-keyword-function atom-function)
```

defines a generic parse function named *parse-function-name*, and a syntactic environment named *environment-name*, in order to parse concrete syntax for *syntax-class-name*. The argument *atom-function* is a parse function which is invoked if the form is not a list, and the argument *no-keyword-function* is a parse function which is invoked if the form is a list but the first element is not a recognised keyword. We can use this form to define `arith-env`.

```
(define-keyword-syntax-class arith-expr
  parse-arith-expr arith-env
  (λ (env form)
    (error form "Could not parse arithmetic expression"))
  (λ (env form)
    (cond ((symbol? form) (object arith-var form))
          ((integer? form) (object arith-const form))
          (else (pass)))))
```

Note that we are using lazy delegation for a new purpose here. Previously, we've been using it to layer abstract syntax; here, we are using it to layer syntactic environments.

Parse functions for non-terminals, such as `parse-arith-add`, typically work by matching a pattern, recursively parsing subforms, and creating an object. Ziggurat provides a utility to automate this process: the macro `parse-pattern-variables`. This section of code, for example, defines a function `parse-arith-add` that matches the concrete syntax of the addition non-terminal and produces the corresponding piece of abstract syntax.

```
(define parse-arith-add
  (parse-pattern-variables '(+ ,parse-arith-expr
                               ,parse-arith-expr)
                           (_ l r)
                           (object arith-add l r)))
```

The Ziggurat macro `parse-pattern-variables` takes three arguments: a pattern, a pattern of variables, and an action. The macro produces a parser that matches the pattern, binds the variables to the sub-forms of the input, and then performs the action.

The expression (`parse-pattern-variables` *P V A*) evaluates to a parse function *R*. Calling (*R env F*) matches the form *F* to the pattern *P* and binds the variable pattern *V* in the action *A*.

- If P is a parser function, it is used to parse the term F; the result value is bound to identifier V, unless V is `_`, the "don't-care" symbol. The action A occurs in the scope of the V binding.
- If P is a symbol, then the pattern matches when the term F is the same symbol. V should again be `_`, or an identifier that will be bound to F.
- If P is a list of sub-patterns $(p_1\ p_2 \ldots p_n)$, and V is a list of variable sub-patterns $(v_1\ v_2 \ldots v_n)$, then a successful pattern-match requires that F be a list of sub-forms $(f_1\ f_2 \ldots f_n)$, and for all $1 \leq i \leq n$, p_i matches f_i and binds the variable pattern v_i.

If the pattern does not match the form, the parser function signals an error.

Once we have defined parse functions for the language primitives, we must define an initial syntactic environment for the language containing those primitives. The macro `make-keyword-syntax-env` instantiates an environment of a class previously declared with `define-keyword-syntax-class`. Parsers are added to an environment with the generic function `add-keyword-parser!`, *e.g.*:

```
(define initial-arith-env (make-keyword-syntax-env arith-env))
(add-keyword-parser! initial-arith-env '+ parse-arith-add)
```

This is all of the low-level machinery necessary to define parsers in Ziggurat. However, we can provide high-level machinery on top of this to make basic language implementation easier.

8.2 Template environments

A macro needs to define two things: a parse function, and a rewrite function. We have mentioned that in Ziggurat, the rewrite function takes the form of the delegate-instantiation method of the new syntax node. One way of designing these methods is to completely instantiate the AST of the rewritten syntax. Doing this with the `arith-sqr` syntax class looks like

```
(define-class arith-sqr (expr)
  (λ ()
    (let ((temp-var-name (gensym))) ; create fresh var
      (object arith-let `(,(object arith-var temp-var-name))
                        `(,expr)
              (object arith-mul
                      (object arith-var temp-var-name)
                      (object arith-var temp-var-name))))))
```

We can simplify this process by using the fact that we already have a parse function handy. Ziggurat's `parse-template-with-variables` macro takes advantage of this. When a `parse-template-with-variables` form is evaluated, it creates a *template environment*. A template environment is a syntactic environment that defines a number of meta-variables. The methods of a template environment are all of the parse functions known to Ziggurat. These parse functions all behave the same way: if the form they are parsing is one of the meta-variables of the template environment, they return the AST bound to it in the template environment. Otherwise, they delegate to the higher-level environment. Invoking (`parse-template-with-variables` e p ((v_1 f_1) ...) g) builds a template environment in which the template variable v_1 is bound to f_1 and so forth. Like other syntactic environments, template environments have enclosing environments; this is the argument e to `parse-template-with-variables`. The form then parses g with the parse function p in the context of this template environment. A simple implementation of a template environment would be:

```
(define-class template-env (variables super)
  (λ () super))

(define-method template-env parse-arith-expr
  (λ (env form)
    (if (symbol? form)
        (let ((match (assq form variables)))
          (if match (cdr match) (pass)))
        (pass))))
```

This allows us to build delegate-instantiation functions out of templates, rather than Ziggurat primitives. If we wished to rewrite the `arith-sqr` delegate-instantiation method via templates, for example, we would write

```
(define-class arith-sqr (expr)
  (λ ()
    (parse-template-with-variables top-level-arith-env
      parse-arith-expr
      (('e expr))
      '(let ((temp e)) (* temp temp)))))
```

This has the advantage of allowing us to write delegate-instantiation methods in our target language, thereby abstracting away from Ziggurat primitives.

$d \in$ DSyn ::= `(define-syntax` *n t*`)`
$e \in$ Expr ::= ...
| `(let-syntax ((`*n t*`) ...)` *e*`)`
$t \in$ Transformer ::= `(syntax-rules (`*n* ...`) ((`*p e*`) ...))`
$p \in$ Pattern ::= `(`*n* `(`*n u*`) ... [...])` ; Optional "..." is literal keyword.
$u \in$ Parser ::= *n*
$n \in$ Name ::= x, y, z, ...

Fig. 5. A grammar for high-level macros

8.3 High-level macros

Now that we have the ability to write the rewrite function in the target language, we need not use the Ziggurat language at all when designing macros. Instead, we can have forms in our target language that define a macro. Ziggurat makes it straightforward to introduce a `let-syntax` form and a `define-syntax` form to the target language, similar to how Figure 5 extends the Arith language. These language additions are a general template for high-level macros; they need not be the only way high-level macros are added to a target language.

The `let-syntax` form defines new syntax keywords, and then parses an expression in the context of those keywords. The `define-syntax` form behaves the same way, but defines the new syntax at the top level, and does not yet parse a new form, *e.g.*

```
(define-syntax square-five
  (syntax-rules ()
    ((square-five) (* 5 5))))
```

This code defines a parse function and a rewrite function. The parse function will create an object of class `extended-syntax`. Each `extended-syntax` object has three fields:

- A field indicating which `syntax-rules` pattern was matched
- A field indicating what variables were bound when the form was parsed
- A rewrite function.

The actual delegate-instantiation method of an `extended-syntax` object merely passes off to the field `rewrite-function`.

```
(define-class extended-syntax
              (matched-pattern variables rewrite-function)
  (λ () (rewrite-function matched-pattern variables)))
```

The field `rewrite-function` is a function defined by the transformer part of the high-level-macro. For the `square-five` example, the value of this field would be:

```
(λ (matched-pattern variables)
  (parse-template-with-variables top-level-arith-env
    variables
    '(* 5 5)))
```

The parse function for `square-five` is simple: it matches only `(square-five)`, parses no sub-expressions, and binds no variables. However, usually we will need to parse sub-forms. Thus, our high-level macro language allows us to parse patterns. We allow only one kind of pattern: a list of sub-forms. Since there are several parse functions one could use to parse a sub-form, we require the macro writer to specify which one.

```
(define-syntax sqr
  (syntax-rules ()
    ((sqr (x arith-expr)) (* x x))))
```

The macro `sqr` has one sub-form. When the parser encounters source code of the form `(sqr 3)`, it first parses `3` as an `arith-expr`, and then binds the parsed form to the variable `x` in the `variables` field of the associated `extended-syntax`.

The list of sub-forms can be made expandable, through the `...` keyword.

```
(define-syntax mult
  (syntax-rules ()
    ((mult (x arith-expr)) x)
    ((mult (x arith-expr) (y arith-expr) ...)
     (let ((temp (mult y ...)))
       (* x temp)))))
```

The variable `y` here is associated with the list of `arith-exprs` that begins with the second sub-form of the `mult` statement, and ends at the end of the form.

The first subform of `syntax-rules` is the set of names we wish to capture. That is a facet of the hygiene system: it means that, when we refer to these names in the macro definition, we wish them to refer to the same things the names would refer to in the context of the macro use. To explain how this works, first we need to explain how hygiene works in Ziggurat.

8.4 Hygiene and referential transparency

Scheme's macro system has a property that we'd like to ensure: we want to make sure that the names introduced and used inside of a macro do not interfere with those appearing where the macro is used.

Consider the arithmetic form `(let ((temp 3)) (mult temp 6))`. A naïve implementation of the `mult` macro presented at the end of the last section would simply rewrite the `mult` statement directly, resulting in the code `(let ((temp 6)) (* temp temp))`. The result of executing this is `36`, instead of the expected `18`. The clear problem here is that the `temp` referred to by the macro writer is not the same `temp` that the macro reader intended to use, but since they have the same name, they are not distinguished. In other words, the macro *captures* the variable `temp`, although we do not want it to.

Ziggurat employs a traditional solution (Dybvig *et al.*, 1992) to this problem, tailored to use lazy-delegation objects and to be language independent. We replace simple Scheme data with *annotated* data, which maintains information about the syntactic context in which it appears:

```
(define-class annotated-datum
  (datum namespace start-location end-location))
```

The `start-location` and `end-location` fields contain source-file information and are merely for informed error-reporting. The `namespace` field is what provides our hygiene functionality. Hygiene comes from the `annotated-datum-eq?` predicate.

```
(define-generic (annotated-datum-eq? a b) #f)

(define-method annotated-datum annotated-datum-eq?
  (λ (a b) (annotated-datum-namespace-eq? b datum namespace)))

(define-generic (annotated-datum-namespace-eq? a dat ns) #f)

(define-method annotated-datum annotated-datum-namespace-eq?
  (λ (a dat ns)
    (and (eq? datum dat)
         (eq? namespace ns))))
```

In the example given above, the two `temp` variables will not collide assuming that the `namespace` fields of these symbols are set correctly. The `namespace` field should properly refer to a lazy-delegation object of class `namespace`. Objects of class `namespace` remember what names they define, and are layered. Symbols can be looked up in namespaces: if a namespace is questioned about a symbol it defines, it returns itself, and if it is questioned about a symbol defined in a higher space, it returns that space. We also have a `role` argument that reflects the syntactic category of the name. This is so that we can maintain different namespaces for different types of syntax nodes.

```
(define-class namespace (super table)
  (λ () super))

(define (new-namespace super)
  (object namespace super (make-hash-table)))

(define-generic (namespace-declare ns role name))

(define-method namespace namespace-declare
  (λ (ns role name)
    (hash-table-put! table role name ns)))

(define-generic (namespace-lookup ns role name) #f)

(define-method namespace namespace-lookup
  (λ (ns role name)
    (hash-table-get table role name pass)))
```

Now, all that needs to be done is to associate symbols with their proper namespace information. Adding namespace information is a form of parsing: when we want to parse

a symbol in a particular context, we will want to maintain that context as part of its semantic information. We therefore add a method `get-current-namespace` to syntactic environments.

```
(define-class namespace-env (namespace super)
  (λ () super))

(define-generic (get-current-namespace env) top-level-namespace)

(define-method namespace-env get-current-namespace
  (λ (env) namespace))
```

It's now a simple matter to alter the definition of `parse-template-with-variables` such that for each template environment, a new namespace is defined.

However, sometimes we want to have variable capture. For these instances, we introduce a capture mechanism into `syntax-rules`.

```
(define-generic (namespace-declare-capture ns role
                                           name captured-ns))

(define-method namespace namespace-declare-capture
  (λ (ns role name captured-ns)
    (hash-table-put! table role name captured-ns)))
```

Thus, when we use `syntax-rules` in a high-level macro, for each name in the argument to `syntax-rules`, we first look it up in the context of the macro use, and put that role into the namespace of the rewrite function.

8.5 Macro-defining macros

Like Scheme, Ziggurat enjoys the powerful feature of macros that define other macros. In fact, since high-level macros utilise parse functions that may well already contain a macro facility, macro-defining macros are included for free. However, in some instances, some additional annotation is required.

Consider the arithmetic macro:

```
(define-syntax macro-with-a-macro
  (syntax-rules ()
    ((macro-with-a-macro (exp arith-expr))
     (let-syntax ((inner-syntax (syntax-rules ()
                                  ((_) 3))))
       (+ exp (inner-syntax))))))
```

Despite the fact that this macro has a macro definition inside of it, it does not require any special techniques to deal with, since it applies the parse function to make a rewrite function. Ziggurat handles this sort of situation automatically.

However, a common use of macro-defining macros is to define a macro for use by the user somewhere else. Let us alter the definition of the above macro slightly:

```
(define-syntax let-three-macro
  (syntax-rules (three)
    ((let-three-macro (exp arith-expr))
     (let-syntax ((three (syntax-rules ()
                           ((_) 3))))
       (exp)))))
```

Now, let's use this new macro:

```
(let-three-macro (+ 7 (three)))
```

Here, when we use the name `three`, we are using it as the keyword for a macro that rewrites to 3. Ziggurat, as specified above, can not handle this case, at least not with high-level macros. When the expression `(+ 7 (three))` is parsed, the macro `three` is not yet defined, since that macro is only defined in the rewrite function of `let-three-macro`. For the writer of a low-level macro, there are two options: either we define the macro `three` earlier, or hold off the parsing of `(+ 7 (three))` until later.

We can build this later method into the high-level macro system directly by use of the `delay` keyword.

```
(define-syntax let-foo-macro
  (syntax-rules (foo)
    ((let-foo-macro (exp arith-expr delay))
     (let-syntax ((foo (syntax-rules ()
                         ((foo (y arith-expr)) y))))
       exp))))
```

The `delay` keyword tells the parse function to delay parsing the `exp` sub-form until it is needed by the rewrite function. Thus, parsing `(let-foo-macro (foo (+ 7 5)))` will parse `(foo (+ 7 5))` in a context where the `foo` macro is defined.

The `delay` macro combines in an interesting way with the namespace mechanism. It is important that the names in the delayed sub-form be parsed in the namespace they originally appear in. Thus, instead of parsing the sub-form, we reify it, and *force* it when it appears in the rewrite function.

```
(define-class delayed-syntax (form namespace))

(define-generic (force-delayed-syntax ds env parse-function))

(define-method delayed-syntax force-delayed-syntax
  (λ (ds env parse-function)
    (parse-function (object namespace-env namespace env) form)))
```

9 An example: assembly language

Every tower must have a ground floor: for our running example, we start our language tower with an assembly language. Our assembly language has a slight twist: in order to make macro writing more convenient, code labels have nested, lexical scope, based on elements we explored in a previous language design (Shivers, 2005).

```
i ∈ Identifier ::= a | b | c | ...
r ∈ Reg        ::= i
l ∈ Label      ::= (star i)
c ∈ Const      ::= ... | -2 | -1 | 0 | 1 | 2 | ...
e ∈ Expr       ::= r | l | c
s ∈ Stmt       ::= (mv r e)
                 | (add r e e)
                 | (ld r e)
                 | (st e e)
                 | (bez e e)
                 | (jmp e)
                 | (let ((l s)...) s)
                 | (letrec ((l s)...) s)
g ∈ Seg        ::= (code s)
                 | (null-segment)
```

Fig. 6. A grammar for a sexp-based assembly language

```
(define-class asm-register (name))
(define-class asm-constant (value))
(define-class asm-label (name))
(define-class asm-mv (dst src))
(define-class asm-add (dst srcl srcr))
(define-class asm-ld (dst src))
(define-class asm-st (dst src))
(define-class asm-bez (tst dst))
(define-class asm-jmp (dst))
(define-class asm-let (labels stms stm))
(define-class asm-letrec (labels stms stm))
(define-class asm-code-segment (stm))
(define-class asm-null-segment ())
```

Fig. 7. Assembly-language abstract syntax as Ziggurat class declarations

9.1 Syntax and dynamic semantics

The grammar of our assembly language is shown in Figure 6. Implementing this in Ziggurat is again a matter of directly transcribing the grammar, as seen in Figure 7. Once again, in the AST node for `let` statements, we separate the labels and the code points they bind.

Expressions specify values that are available without computation. We have three forms of expression.

- Constants c are data values encoded directly in the program. Values in assembly are fundamentally untyped; the only way to distinguish an integer from a floating-point value from a data pointer from a code pointer is by context.
- Registers i are the variables of this language level. Registers are not declared; the value of a register is set via assignment, and the value of a register is undefined before it is assigned for the first time. At this language level, we have an unlimited number of registers.
- Labels `(star` i`)` represent program locations. As a convenience, the S-expression

reader interprets `*`*i* as `(star foo)`, similar to the way Scheme readers handle the `'foo` form as `(quote foo)`. Labels are bound to code points by the `let` and `letrec` instructions, and are first-class values: they can be assigned to registers, stored in memory, and so forth.

Statements manipulate registers, perform control transfers, and manipulate the store. We have eight forms of statement.

- `(mv` *r e*`)` moves the value specified by *e* into the register *r*, and then branches to the label `*next` in the scope of the statement.
- `(add` *r* e_1 e_2`)` adds the value specified by e_1 to e_2, and stores the result in register *r*. Then, it branches to the label `*next`.
- `(ld` *r e*`)` loads the value in the store at location *e*, and stores it in the register *r*. Control then branches to `*next`.
- `(st` e_1 e_2`)` stores the value e_2 at location e_1 in the store, and then branches to `*next`.
- `(bez` e_1 e_2`)` tests value e_1. If it is equal to zero, then control branches to the code at location e_2. Otherwise, control branches to the label `*next`.
- `(jmp` *e*`)` always transfers control to *e*.
- `(let ((`l_1 s_1`) (`l_2 s_2`) ...)` s_b`)` sets up a context in which the label l_1 is bound to the statement s_1, l_2 to s_2 and so forth, and then evaluates s_b in this context. These bindings for $l_1, l_2, \ldots$ are not available in $s_1, s_2, \ldots$
- `(letrec ((`l_1 s_1`) (`l_2 s_2`) ...)` s_b`)` works similarly to `let`, with the important distinction that labels $l_1, l_2, \ldots$ are available in the scope of $s_1, s_2, \ldots$

Our assembly language also has segments. A segment represents a whole program.

- `(code` *s*`)` represents a block of code.
- `(null-segment)` is an empty segment of code.

The `let` and `letrec` instructions are the only way in this language to combine and order instructions. This means that programs must be written, essentially, in reverse, with later instructions occurring earlier in the program text. This can make assembly language programs difficult to read. Here, for example, is a statement to multiply 3 by 5:

```
(let ((*exit (jmp *next)))
  (letrec ((*loop (let ((*next (jmp *loop)))
                    (let ((*next (add i i -1)))
                      (let ((*next (add y y x)))
                        (bez i *exit))))))
    (let ((*next (jmp *loop)))
      (let ((*next (mv y 0)))
        (let ((*next (mv i 5)))
          (mv x 3) )))))
```

Clearly, some syntax extensions to this core language are needed in order to make it possible to write comprehensible programs.

```
s ∈ Stmt        ::= ...
                  | (let-syntax ((n t) ...) s)
g ∈ Seg         ::= ...
                  | (define-syntax n t)
t ∈ Transformer ::= (syntax-rules (n ...) ((p s) ...))
p ∈ Pattern     ::= (n (n u) ... [...])     ; Optional "..." is literal keyword.
u ∈ Parser      ::= asm-exp | asm-stm
```

Fig. 8. A grammar for assembly macros

9.2 Some simple macros

Writing code at this low level can be difficult. The core assembly language was designed with two goals in mind:

- to have locally scoped labels, to aid in writing macros, and
- given this, to make the language as syntactically simple as possible.

There is no reason at the lowest assembly language for there to be any conveniences to the programmer to make code easier to write and to read, since these can be easily added as macros.

To begin with, the way of sequencing instructions is awkward. What we would like would be a way to put instructions in sequential order. Instead of writing

```
(let ((*next (add z x y)))
  (let ((*next (mv y 5)))
    (mv x 3)))
```

we would like to be able to write

```
(seq (mv x 3)
     (mv y 5)
     (add z x y))
```

In order to do this, we have two ways to define high-level macros: one global, and one local, as seen in Figure 8. This macro system is a default implementation of the one presented in Section 8.

Using this macro system, we can define a `seq` macro:

```
(define-syntax seq
  (syntax-rules ()
    ((seq (s1 asm-stm)) s1)
    ((seq (s1 asm-stm) (s2 asm-stm) ...)
     (let ((*next (seq s2 ...)))
       s1))))
```

Another kind of macro we might want to write would provide more advanced control-flow. Writing a loop, as we see above, is very awkward in the base assembly language. What we would like would be a simple `loop` keyword, written `(loop` *e s*`)`, that would execute the statement *s* for *e* iterations. This can be done with high-level macros.

```
(define-syntax loop
  (syntax-rules ()
    ((loop (n asm-exp) (s asm-stm))
     (letrec ((*loop (seq (bez loopvar *escape)
                          s
                          (add loopvar loopvar -1)
                          (jmp *loop)))
              (*escape (jmp *next)))
       (seq (mv loopvar n)
            (jmp *loop))))))
```

With these macros, we can now rewrite the multiplication example in the previous section:

```
(seq (mv x 0)
     (loop 5
       (add x x 3)))
```

9.2.1 High-level assembly macros and namespaces

The namespace mechanism described in the last section works well with the assembly language. Labels work with it directly. When a label is declared, we merely need to call `namespace-declare` on the current namespace, and when we parse a label, we merely need to call `namespace-lookup`. There is only one exception to this rule, and that is `next`. The label `next` is always implicitly captured: it exists at the top-level assembly namespace only. When `let` and `letrec` statements bind the label `next`, the label is found in the top-level namespace.

Registers would seem to be incompatible with namespaces, since they are not declared. However, we can introduce a rule: if a register is introduced via the hygiene-capturing feature of `syntax-rules`, it is declared; otherwise, registers in separate namespaces are always separate. Thus, in the statement

```
(let-syntax ((init-i (syntax-rules () ((init-i) (mv i 40))))
             (dec-i  (syntax-rules ()
                       ((dec-i)  (add i i -1)))))
  (seq (init-i)
       (loop i (dec-i))))
```

the occurrences of the name `i` in `init-i`, `dec-i` and the `loop` statement are implicitly referring to separate registers, whereas in the statement

```
(let-syntax ((init-i (syntax-rules (i) ((init-i) (mv i 40))))
             (dec-i  (syntax-rules (i)
                       ((dec-i)  (add i i -1)))))
  (seq (init-i)
       (loop i (dec-i))))
```

the four occurrences of `i` are referring to the same register. It is interesting to note that we have allowed separate hygiene rules for separate sorts of identifiers.

10 Termination analysis

An observation we frequently would like to make on our code is whether it terminates or not. This observation is often useful to know just for itself—we would often like to ensure that code we write will not lock up under any circumstances. Additionally, this observation is required for some optimisations, such as constant folding. We provide this capability by implementing a basic analysis for the core language, and permitting it to be extended for embedded languages that might have more restricted semantics.

Our analysis takes the form of a generic function `halts?`. If `halts?` returns a true value for a syntax node, then the analysis has determined that control will exit that node in all environmental contexts. If there exists a context under which control will never exit the specified syntax node, then the method will return `#f`. If the analysis cannot determine whether or not the current node will terminate, it will return `#f`. In this way, the analysis is conservative.

10.1 A simple algorithm for termination analysis

At the assembly language level, we would prefer not to do a complicated analysis; most of our information will come from higher language levels. So, our algorithm at this language level is very basic. The method `halts?` on `letrec` nodes always returns `#f`. Additionally, `jmp` nodes always return `#f`, not because they do not terminate, but instead because statements containing them are not guaranteed to terminate, since it is difficult to track where control goes. Applying `halts?` to a `let` node returns `#t` if and only if `halts?` produces true when applied to every sub-expression of the `let` node. Here is the `halts?` method for `let` nodes:

```
(define-method asm-let halts?
  (λ (s) (and (halts? stm)
              (andmap halts? stms))))
```

Note that every assembly-language statement containing a loop or a recursive call will necessarily return `#f`.

10.2 Extending termination analysis

In its current incarnation, this algorithm is not very useful. The generic function `halts?` will return `#f` for code of any significant complexity. As we move up the language tower, we will have much more interesting things to say about termination analysis. Currently, for example, calling `halts?` on the multiplication example presented above will return `#f`, since the rewritten expression is a `letrec` node.

With the language extensions we have introduced to the assembly language, we already have a good opportunity for improving the analysis: the `loop` macro is a good source of control information. In order to provide a more precise analysis for `loop` nodes, we merely override the `halts?` method for loops:

$$
\begin{array}{lll}
\tau \in \text{TypeSchema} & ::= & \forall \overline{v}.ct \\
v \in \text{TypeVar} & ::= & \alpha, \beta, \gamma, \ldots \\
t \in \text{Type} & ::= & \mathbf{word} \mid ct \mid v \\
ct \in \text{CodeType} & ::= & (r : t; ct) \\
 & & \mid v
\end{array}
$$

Fig. 9. A type system for the assembler language

```
(define-method loop halts?
  (λ (s) (and (asm-constant? n)
              (<= 0 (asm-constant-value n))
              (halts? s))))
```

The analysis is now more advanced. A loop node is now defined to halt whenever its loop bound is a constant, the bound is at least `0`, and the loop body halts. Now, running `halts?` on the multiplication example will return `#t`.

11 A type system for assembly language

An analysis we would like to be able to do at the assembly-language level is type analysis. There are several purposes that a typed assembly language such as the one developed by Morrisett *et al.* (1999) can fulfill, but our main focus here will be on debugging: we would like our type system to catch errors arising from misuse of data. It would be possible for such a type system to make stronger guarantees about the code it is used to analyze, and to implement this system in Ziggurat; however, such a system would require a radically different design than is presented here.

For the assembly language, both statements and expressions are typed. The type of an expression represents the kind of value it produces, while the type of a statement represents requirements on registers on entry to that statement. In order for this to work, we need parametric polymorphism: although `(add z x y)` requires `x` and `y` to be numbers, `z` can have any type on entry, since its value will be overwritten. We refer to the types of statements as *code types.* Since labels are bound to statements, and labels can be used as expressions, expressions can have code types, as can registers: this reflects the fact that registers can hold code pointers.

At this language level, we have only one base data type: **word**. A **word** can be an integer, a character, or a pointer to structured data. This part of the type system is deliberately kept open, with the expectation that higher language levels will elaborate on it.

A monomorphic code type would be a mapping from all possible registers to types. Since this would be impractical to represent, and also too restrictive, we employ polymorphism by means of the type system in Figure 9. Using this type system, we can specify the code type that only requires the register `r` to have type **word** as $\forall\alpha.(\mathbf{r} : \mathbf{word}; \alpha)$.

Using this basic system, we can define three typing relations: one for segments, one for statements, and one for expressions. For segments, the relation $g\bot$ indicates that the segment g has a correct typing. For statements, the relation $E \vdash s : ct$ specifies that in label

$$\frac{E \vdash e : t \text{ in } ct \quad E \vdash r : t \text{ in inst}(E[\texttt{next}]) \quad \text{inst}(E[\texttt{next}])/r = ct/r}{E \vdash [\![\texttt{(mv } r\ e\texttt{)}]\!] : ct}\ \textsc{MvType}$$

$$\frac{E \vdash e_l : \textbf{word} \text{ in } ct \quad E \vdash e_r : \textbf{word} \text{ in } ct \quad E \vdash r : \textbf{word} \text{ in inst}(E[\texttt{next}]) \quad \text{inst}(E[\texttt{next}])/r = ct/r}{E \vdash [\![\texttt{(add } r\ e_l\ e_r\texttt{)}]\!] : ct}\ \textsc{AddType}$$

$$\frac{E \vdash e : \textbf{word} \text{ in } ct \quad E \vdash r : t \text{ in inst}(E[\texttt{next}]) \quad \text{inst}(E[\texttt{next}])/r = ct/r}{E \vdash [\![\texttt{(ld } r\ e\texttt{)}]\!] : ct}\ \textsc{LdType}$$

$$\frac{E \vdash e' : t \text{ in } ct \quad E \vdash e : \textbf{word} \text{ in } ct \quad \text{inst}(E[\texttt{next}]) = ct}{E \vdash [\![\texttt{(st } e\ e'\texttt{)}]\!] : ct}\ \textsc{StType}$$

$$\frac{E \vdash e : ct \text{ in } ct}{E \vdash [\![\texttt{(jmp } e\texttt{)}]\!] : ct}\ \textsc{JmpType}$$

$$\frac{E \vdash e' : ct \text{ in } ct \quad \text{inst}(E[\texttt{next}]) = ct \quad E \vdash e : \textbf{word} \text{ in } ct}{E \vdash [\![\texttt{(bez } e\ e'\texttt{)}]\!] : ct}\ \textsc{BezType}$$

$$\frac{E \vdash s_1 : ct_1, \ldots, E \vdash s_j : ct_j \quad E[l_1 \mapsto \text{gen}(ct_1), \ldots, l_j \mapsto \text{gen}(ct_j)] \vdash s : ct}{E \vdash [\![\texttt{(let ((}l_1\ s_1\texttt{)}\ldots\texttt{(}l_j\ s_j\texttt{)) } s\texttt{)}]\!] : ct}\ \textsc{LetType}$$

$$\frac{E' \vdash s : ct \quad E' \vdash s_1 : ct_1, \ldots, E' \vdash s_j : ct_j}{E \vdash [\![\texttt{(letrec ((}l_1\ s_1\texttt{)}\ldots\texttt{(}l_j\ s_j\texttt{)) } s\texttt{)}]\!] : ct}\ \textsc{LetrecType}$$
$$\text{where } E' = E[l_1 \mapsto \text{gen}(ct_1), \ldots, l_j \mapsto \text{gen}(ct_j)]$$

$$\frac{}{E \vdash c : \textbf{word} \text{ in } ct}\ \textsc{ConstType} \qquad \frac{}{E \vdash l : \text{inst}(E[l]) \text{ in } ct}\ \textsc{LabelType}$$

$$\frac{}{E \vdash r : ct[r] \text{ in } ct}\ \textsc{RegisterType}$$

$$\frac{\vdash s : \alpha}{\texttt{(code } s\texttt{)}\bot}\ \textsc{CodeSegmentType}$$

Fig. 10. Typing relations for the assembly language

environment E, statement s has type ct. A label environment is a partial mapping from labels to type schemas. For expressions, the relation $E \vdash e : t$ in ct specifies that in label environment E, inside of a statement of type ct, an expression e has type t. For example, we might conclude that $\{\texttt{next} \mapsto \forall\alpha.(r_2 : \textbf{word}; \alpha)\} \vdash [\![\texttt{(mv } r_2\ r_1\texttt{)}]\!] : (r_1 : \textbf{word}; \beta)$ holds. The mutually recursive definitions of these relations are shown in Figure 10.

Consider, for example, the MvType rule. We wish to show that $E \vdash [\![\texttt{(mv } r\ e\texttt{)}]\!] : ct$. First, we look up the type of the expression e in ct. This corresponds to the precondition $E \vdash e : t$ in ct, which can be derived from one of the three expression type rules. Next, we require that the destination register have the same type, which corresponds to the precondition $E \vdash r : t$ in inst$(E[\texttt{next}])$. Since we require that the destination register have this type after the instruction has completed, this is a precondition on the type of the `next`

label, which can be found by looking up `next` in the current label environment: $E[\texttt{next}]$. Finally, we require that registers not altered by the `mv` instruction retain their type. This corresponds to $\text{inst}(E[\texttt{next}])/r = ct/r$. The rules for `add`, `ld` and `st` are analogous.

A word about the "function" inst(): this is the instantiation relation—it is not technically a function, despite the fact that we use function notation. When we write $\text{inst}(\tau) = ct$, we mean that the pair (τ, ct) is in the inst relation, *i.e.*, that ct instantiates type schema τ. When we write that $\text{inst}(\tau)/r = ct/r$, we mean that the condition on $\text{inst}(\tau)$ is satisfied by *some* ct' that instantiates τ. In more mathematical terms, $\exists ct' .\ ct'$ instantiates τ and $ct'/r = ct/r$.

The rules for `jmp` and `bez` are not complex, despite their control effects. If we have an instruction `(jmp` e`)`, we merely check that the type of the expression e is compatible with the type of the entire statement. The rule for `bez` is analogous.

The rules for `let` and `letrec` rely on the generalisation function gen(), which takes a type and returns a type schema. It does this by finding all of the type variables in the type, replacing them with fresh type variables, and quantifying over those variables. The gen() function takes an extra argument, which we have elided for brevity's sake. Although its behavior is predictable, it requires fresh type variables, which must be different for each invocation. Some additional machinery is necessary to make this happen, for example, by threading a state through the type computation, but the specifics are not important to this paper.

11.1 Type reconstruction via Kanren

For our type system to act as a debugger, we need some way to assign types to syntax nodes. Ziggurat gives us an opportunity to build an extensible algorithm for this. For each syntax node, we generate a constraint. Then we solve the resulting system of constraints in order to come up with a correct type assignment. When we generate new classes of syntax object, we (optionally!) override the constraint-generating function as needed. This is how we allow global analyses in an environment where methods can only manipulate local data. In order to solve these constraints, we employ a unification-based constraint-solver called Kanren (Byrd & Friedman, 2006), modified to employ lazy delegation.

Kanren works by solving goals. A Kanren goal is a statement of relations between structures, either of which may contain variables. Solving the goal consists of either finding a mapping of variables to values that makes the relations true, or discovering that no such mapping exists. The basic goal in Kanren is unification. By specifying `(== a b)`, we assert that the structures `a` and `b` are the same. In the original Kanren, `a` and `b` are Scheme data, but for our purposes, we will need a bit more complex structure. We require that `a` and `b` be lazy-delegation objects.

Just as we have three varieties of type in the type system, we have three classes of type in the Kanren implementation. Our three classes are variables, the **word** type, and label types. However, we differentiate between label types and partially-instantiated row types, and represent the latter as its own class:

```
(define-class word-type ())
(define-class label-type (row))
(define-class row-type (name type rest))
```

Type variables are special. We use Kanren's built-in variables for these. They are introduced by the `fresh` macro:

```
(fresh (alpha beta) ...)
```

In addition, we have a class to represent type schemas:

```
(define-class forall-type-schema (vars type))
```

The purpose of representing these as lazy delegation objects is to allow us to write specialised unification methods. In the basic Kanren, since we are unifying basic Scheme list structure, when we assert that `(== '(a . b) '(c . d))`, it is assumed that this implies `(== a c)` and `(== b d)`. However, this does not apply to row types. If we have the row type $(\mathtt{x} : \alpha; \beta)$ and we wish to unify it with the row type $(\mathtt{x} : \gamma; \delta)$, we unify, as sub-goals, α with γ, and β with δ. To unify $(\mathtt{x} : \alpha; \beta)$ and $(\mathtt{y} : \gamma; \delta)$, the process is only slightly more complex: we generate a fresh logic variable ϵ, and unify β with $(\mathtt{y} : \gamma; \epsilon)$, and δ with $(\mathtt{x} : \alpha; \epsilon)$. We can accomplish this by making `unify` a generic function on type objects, and directly encoding these rules as Kanren goals.

```
(define-method row-type unify
  (λ (v w) (unify-row w name type rest)))

(define-generic (unify-row v name-prime type-prime
                           rest-prime))

(define-method row-type (unify-row v name-prime type-prime
                                   rest-prime)
  (if (eq? name name-prime)
      (all (== type type-prime)
           (== rest rest-prime))
      (fresh (new-row-var)
        (== rest (object row-type name-prime
                         type-prime new-row-var))
        (== rest-prime (object row-type name
                               type new-row-var))))))
```

Now that we have types representable in Kanren, it's a fairly simple matter to encode the type rules as unification constraints. For each syntax class, we write a method `(make-stm-type-goal s ct e)`, that returns a goal asserting that a statement `s` has type `ct` in label environment `e`.

For example, take the MVTYPE rule.

$$\frac{\begin{array}{c} E \vdash e : t \text{ in } ct \\ E \vdash r : t \text{ in } \mathrm{inst}(E[\mathtt{next}]) \\ \mathrm{inst}(E[\mathtt{next}])/r = ct/r \end{array}}{E \vdash [\![(\mathtt{mv}\ \ r\ \ e)]\!] : ct}\ \textsc{MvType}$$

The generic function `(make-stm-type-goal s ct e)` is the implementation of $E \vdash s : ct$. Therefore, the goal that `make-stm-type-goal` returns must reflect each of the

preconditions of this type rule. The `make-stm-type-goal` method for a `mv` syntax object is:

```
(define-method asm-mv make-stm-type-goal
  (λ (s ct e)
    (let* ((next-type (instantiate-type
                       (type-env-lookup e 'next))))
      (fresh (after-dst-type before-dst-type rest-type)
        (make-exp-type-goal src after-dst-type e ct)
        (fresh (rest)
          (== ct (object label-type
                         (object row-type (asm-var-name dst)
                                          before-dst-type
                                          rest-type)))
          (== next-type
              (object label-type
                      (object row-type (asm-var-name dst)
                                       after-dst-type
                                       rest-type)))
```

To understand how this method works, we take it line by line. First, we must define a type to represent the type of the label `next`. This is done by locally binding `next-type` to the result of a call to `instantiate-type`. The function `instantiate-type` takes a type schema, and replaces each of its type variables with fresh Kanren logic variables. This corresponds to inst() in our type system. Thus, the local variable `next-type` holds the type of the label `next`.

Next, we create three fresh logic variables with the `fresh` form: `after-dst-type`, `before-dst-type` and `rest-type`. If the `mv` statement is of the form `(mv` *dst src*`)`, then `after-dst-type` is the type of *dst* after the move instruction, `before-dst-type` is its type before, and `rest-type` represents the types of all registers not affected by the move.

We need to assert that the type of the source expression is the same as the destination register's type, `dst-type`. This corresponds to the precondition $E \vdash e : t$ in *ct*. This is accomplished by the goal constructed by generic function `(make-exp-type-goal` *e t E ct*`)`. The `make-stm-type-goal` method next performs two unifications, which form the meat of the type rule's implementation:

- The first unification asserts that the statement's type maps the destination register to `before-dst-type` and the rest to `rest-type`; *i.e.*, that its type (in the syntax of the type system) is $(\textit{name} : \textit{before-dst-type}; \textit{rest-type})$.
- The second unification asserts that the type of the next instruction to be executed is $(\textit{name} : \textit{after-dst-type}; \textit{rest-type})$.

The rules for `add`, `jmp` and `bez` are similar. However, with `let` and `letrec`, the story is a bit more complicated. In this case, we have to gather up the constraints for each of the sub-expressions, solve them, and assign them to variables in order to build a new label environment. In order to do this, we need a generalisation function.

```
(define (generalize-type t)
  (object forall-type-schema (find-variables t) t))
```

The generalisation function takes a type, and returns a type schema, quantified over the variables found therein. In other type-reconstruction algorithms, the generalisation function must take another argument, representing variables *not* to generalise; however, in our system we do not have function definitions or let-bound variables, so this is not an issue. In Kanren, we run (that is, solve) goals with the `run` macro. Therefore, our `make-stm-type-goal` for `asm-let` looks like this:

```
(define-method asm-let make-stm-type-goal
  (λ (s ct e)
    (let ((new-env
           (let loop ((labels labels) (stms stms) (env e))
             (if (pair? labels)
                 (loop (cdr labels) (cdr stms)
                       (extend-environment
                          env
                          (asm-label-name (car labels))
                          (generalize-type
                             (run 1 (t)
                                  (make-stm-type-goal
                                    (car stm) ct e))))))
                 env)))
      (make-stm-type-goal stm t new-env))))
```

The argument `1` to `run` specifies that we want at most one solution.

11.2 Extending type judgements for new syntax

Since type goals are generic functions on syntax nodes, they interact transparently with macros. For example, if we were to type-check the `loop` macro introduced above, it would work with our type system with no extension. If we were to typecheck the erroneous statement

```
(loop *next (add x x 1))
```

Because there is no `make-stm-type-goal` method defined on `loop`, Ziggurat would apply the delegation method and rewrite the statement to

```
(letrec ((*loop (seq (bez loopvar *escape)
                     (add x x 1)
                     (add loopvar loopvar -1)
                     (jmp *loop)))
         (*escape (jmp *next)))
   (seq (mv loopvar *next)
        (jmp *loop))))))
```

This would cause a type error, as expected. However, there is a problem with this: although this code generates an error, the statement that is actually causing the problem and thus would be reported by the type-checker is `(mv loopvar *next)`, which does not appear in the original code at all.

For better error-reporting, then, we might want to have a specialised method for invoking `make-stm-type-goal` on `loop` statements. The type judgement for `loop` statements is simple: it merely needs to verify that the loop variable is a **word** type, and that the inner statement type-checks.

```
(define-method asm-loop make-stm-type-goal
  (λ (s ct e)
    (all
       (make-exp-type-goal loop-bound (object word-type) e ct)
       (make-stm-type-goal loop-stm ct e))))
```

Upon defining this, the type-checker will correctly report that the problem is with the loop bound.

11.3 Sum and product types

Our type system has a flaw: it has nothing to say about the state of memory. Therefore, our type goals for `ld` and `st` instructions necessarily require a loss of information. When loading from memory, the type of the loaded value is allowed to have any type.

```
(define-method asm-ld make-stm-type-goal
  (λ (s ct e)
    (let* ((next-type
            (instantiate-type (type-env-lookup e 'next))))
      (fresh (dst-type mirror-dst-type rest-type)
        (make-exp-type-goal src reference-word-type e ct)
        (== ct (object label-type
                       (object row-type (asm-var-name dst)
                                        mirror-dst-type
                                        rest-type)))
        (== next-type
            (object label-type
                    (object row-type (asm-var-name dst)
                                     dst-type
                                     rest-type))))))
```

Because of this permissive type system, it is possible to store a word value to memory, and read it back as a code pointer. However, if we implement structured data with macros, we can extend our implementation of the type system to better aid us in debugging.

At the next level above our core assembly language, we add only three kinds of structured data: sum types, product types, and named recursive types. We will defer discussing named types for the moment. We define the rest of the new grammar in Figure 11.

$s \in$ Stmt ::= ...
| (kons r e e)
| (kar r e)
| (kdr r e)
| (left r e)
| (right r e)
| (branch r l l)

Fig. 11. Extending the assembler syntax to provide sum and product datatypes

- **kons**, **kar** and **kdr** define product types. The (kons r x y) form builds a two-place data structure (or *pair*) containing the values of x and y, and places the address of the pair in r. A (kar r x) form extracts the first value of pair x and places the result in r; likewise, (kdr r x) extracts the second value.
  The macro for kons expands into code that performs a system call to a malloc procedure to allocate a new two-word block of storage; this keeps the macro simple and the extraneous details of memory management off-stage so that we may focus on our static semantics. Figure 12 shows the implementation of the pair operations.
- **left**, **right** and **branch** produce and consume values with sum types. A (left r e) form builds a two-way sum object by injecting the value of e into its left side; likewise, (right r e) injects e into the right side of the sum. A (branch r l_1 l_2) performs a conditional branch based on the value of r: if it was constructed with left, then control transfers to l_1, otherwise, it jumps to l_2. In both cases, r is overwritten with the injected value.

Providing advanced type constructors is not adequate to introduce an advanced type system. Despite the fact that we have built macros for structured data, the core type system lacks the type structure permitting it to discriminate correct uses of structured data from incorrect ones. Code such as

```
(seq (kons x 1 2) (kdr y x) (kdr z y))
```

will not signal an error, as it still delegates handling for type checking to the underlying expanded code. However, we are now in a position to add extension-specific static semantics to the new forms, as shown in Figure 13.

The problem with our type system is that it has no sum types, product types or named types: all of these are represented by the universal **word** type. So, we begin by defining these, as lazy-delegation object classes. There is very little functionality in the basic definition of the classes: at a lower language level, structured data simply looks like a pointer, and the rewrite function reflects that.

To implement the type rules in Figure 13, we must first define our new types.

```
(define-class pair-type kar kdr
  (λ () reference-word-type))
```

```
(define-syntax kons
 (syntax-rules ()
  ((kons (n asm-var) (kar asm-exp) (kdr asm-exp))
   (let ((*k (seq (mv n rv) ; memory location allocated
                  (st n kar)
                  (add tmp n 1)
                  (st tmp kdr))))
     (seq (mv arg1 2)  ; number of words to allocate
          (mv rp *k)   ; return point
          (jmp *malloc))))))

(define-syntax kar
 (syntax-rules ()
  ((kar (x asm-var) (k asm-exp))
   (ld x k))))

(define-syntax kdr
 (syntax-rules ()
  ((kdr (x asm-var) (k asm-exp))
   (seq (add tmp k 1)
        (ld x tmp)))))
```

Fig. 12. Macros for structured data. Note that the identifiers `rv`, `rp` and `arg1` are globally defined at top level as fixed registers.

```
(define-method pair-type unify
  (λ (u v) (unify-kar-kdr v kar kdr)))

(declare-generic (unify-kar-kdr v kar kdr)
  (λ (v kar kdr) (type-error "not a pair type" v)))

(define-method pair-type unify-kar-kdr
 (λ (v kkar kkdr)
   (all (== kar kkar)
        (== kdr kkdr))))
```

We want product types to unify with other product types, and not with any other structured types. Product type s unifies with product type t iff the kar of s unifies with the kar of t, and the kdr of s unifies with the kdr of t. This is directly represented in the code that implements the type rule (Figure 14).

11.4 Adding named types

Now that we have types for structured data, the next language feature we'd like to add is the ability to name types, which is useful, *e.g.*, for recursively defined type structures such as lists and trees. In order to associate a named type with a form, we must create an additional class of syntax node to represent a structured type, and insert these nodes into

$$\frac{\begin{array}{c}E \vdash e_l : t_l \text{ in } ct \\ E \vdash e_r : t_r \text{ in } ct \\ E \vdash e : \mathbf{pair}(t_l, t_r) \text{ in } \text{inst}(E[\texttt{next}]) \\ \text{inst}(E[\texttt{next}])/r = ct/r\end{array}}{E \vdash [\![(\texttt{kons } r \ e_l \ e_r)]\!] : ct} \text{ KONSTYPE}$$

$$\frac{\begin{array}{c}E \vdash e : \mathbf{pair}(t_l, t_r) \text{ in } ct \\ E \vdash r : t_l \text{ in } \texttt{inst}(E[\texttt{next}]) \\ \text{inst}(E[\texttt{next}])/r = ct/r\end{array}}{E \vdash [\![(\texttt{kar } r \ e)]\!] : ct} \text{ KARTYPE}$$

$$\frac{\begin{array}{c}E \vdash e : \mathbf{pair}(t_l, t_r) \text{ in } ct \\ E \vdash r : t_r \text{ in } \texttt{inst}(E[\texttt{next}]) \\ \text{inst}(E[\texttt{next}])/r = ct/r\end{array}}{E \vdash [\![(\texttt{kdr } r \ e)]\!] : ct} \text{ KDRTYPE}$$

$$\frac{\begin{array}{c}E \vdash e : t_l \text{ in } ct \\ E \vdash r : \mathbf{sum}(t_l, t_r) \text{ in } \text{inst}(E[\texttt{next}]) \\ \text{inst}(E[\texttt{next}])/r = ct/r\end{array}}{E \vdash [\![(\texttt{left } r \ e)]\!] : ct} \text{ LEFTTYPE}$$

$$\frac{\begin{array}{c}E \vdash e : t_r \text{ in } ct \\ E \vdash r : \mathbf{sum}(t_l, t_r) \text{ in } \text{inst}(E[\texttt{next}]) \\ \text{inst}(E[\texttt{next}])/r = ct/r\end{array}}{E \vdash [\![(\texttt{right } r \ e)]\!] : t} \text{ RIGHTTYPE}$$

$$\frac{\begin{array}{c}E \vdash r : \mathbf{sum}(t_l \ t_r) \text{ in } ct \\ ct/r = t \\ E \vdash l_l : [r \mapsto t_l]t \text{ in } ct \\ E \vdash l_r : [r \mapsto t_r]t \text{ in } ct\end{array}}{E \vdash [\![(\texttt{branch } r \ l_l \ l_r)]\!] : ct} \text{ BRANCHTYPE}$$

Fig. 13. Type relations for structured types

the language as extensions. This is reflected in the fact that the grammar now requires new syntax categories. The new additions are shown in Figure 15.

- **typedef** introduces named types. The `(typedef ((`n t`)) `s`)` form introduces one named type, n, in the statement s. The type t describes the internal format of objects of type n. The name n is visible in t; this allows the language user to define a recursive type.
- **name** and **unname** construct and deconstruct named types. The `(name `r n e`)` form introduces into register r data of form e, with the named type n; e must have the same type as the form of n. Likewise, `(unname `r n e`)` takes data of named type n from e, strips off the name, and puts the result in r.

```
(define-method kons make-stm-type-goal
 (λ (s ct e)
   (let ((next-type (instantiate-type
                      (type-env-lookup e 'next))))
     (fresh (kar-type kdr-type kons-type
             mirror-type rest-type)
      (make-exp-type-goal kar kar-type e ct)
      (make-exp-type-goal kdr kdr-type e ct)
      (== kons-type (object pair-type kar-type kdr-type))
      (== next-type
          (object label-type
            (object row-type (asm-var-name n)
                             kons-type rest-type)))
      (== ct (object label-type
               (object row-type (asm-var-name n)
                                mirror-type rest-type)))))))
```

Fig. 14. Extending the type system for tuple construction.

$n \in$ TypeName	::=	`a` | `b` | `c` | ...	
$t \in$ Type	::=	`word`	
		| n	; Named-type reference
		| `(x` t t`)`	; Product type
		| `(+` t t`)`	; Sum type
$s \in$ Stmt	::=	...	
		| `(typedef ((`n t`)...)` s`)`	
		| `(name` r n e`)`	
		| `(unname` r n e`)`	

Fig. 15. Extending the assembler syntax to provide named types

This is easily implemented in Ziggurat. First, we must have a class for named types. A named type compiles into its internal form. For example, if named type `inflist` is declared to be `(x word inflist)`, then the object representing the type `inflist` rewrites to a pair-type object containing a `word` and an `inflist`.

```
(define-class named-type (name form)
  (λ () form))
```

Unification for a `named-type` is simple: a named type only unifies with itself.

We still need to define type goals in this augmented type system. The goals we will implement are shown in Figure 16. We introduce the notion of a *type-name environment*. A type-name environment is a mapping from type names to types. A type goal is now of the form $E, F \vdash s : ct$, meaning that in variable environment E and type-name environment F, statement s has type ct. Implementing this would seem to present a complication: the object-oriented methods we use to create type goals take only one environment as an argument. In actuality, this is not a problem: since the environment is a lazy-delegation object, we can layer different forms of type environment in the same way we layer different forms of syntactic environment. By building a named-type environment that delegates to a raw type environment, that environment can serve double duty.

$$\frac{E, F' \vdash s : ct}{E, F \vdash [\![\texttt{(typedef ((}n_1\ f_1\texttt{) ...) }s\texttt{)}]\!] : ct}\ \textsc{TypedefType}$$
where $F' = F[n_1 \mapsto s_1, \ldots]$

$$\frac{\begin{array}{c}E \vdash r : F[n] \text{ in } ct \\ E \vdash e : n \text{ in inst}(E[\texttt{next}]) \\ \text{inst}(E[\texttt{next}])/r = ct/r\end{array}}{E, F \vdash [\![\texttt{(name }r\ n\ e\texttt{)}]\!] : ct}\ \textsc{NameType} \qquad \frac{\begin{array}{c}E \vdash r : n \text{ in } ct \\ E \vdash e : F[n] \text{ in inst}(E[\texttt{next}]) \\ \text{inst}(E[\texttt{next}])/r = ct/r\end{array}}{E, F \vdash [\![\texttt{(unname }r\ n\ e\texttt{)}]\!] : ct}\ \textsc{UnnameType}$$

Fig. 16. Type rules for named types

With this addition, defining type goals is straightforward. Notably, there are no type goals generated by the `typedef` syntax; it merely uses the type goals from its inner statement, after augmenting the type-name environment. The `name` and `unname` statements merely enforce that uses of a named type match the form of that named type, as found in the type-name environment.

```
(define-class asm-name (dst src type)
  (λ () (object asm-mv dst src)))

(define-method asm-name make-stm-type-goal
  (λ (s ct e)
    (let* ((next-type (instantiate-type
                       (type-env-lookup e 'next))))
      (fresh (mirror-dst-type rest-type)
        (make-exp-type-goal dst type e ct)
        (== ct (object label-type
                       (object row-type (asm-var-name dst)
                               mirror-dst-type rest-type)))
        (== next-type
            (object label-type
                    (object row-type (asm-var-name dst)
                            type rest-type)))))))
```

Named types require us to write new parsing methods. We now have a new kind of syntax for type forms, which means we need new syntax classes, syntax environments, and parse methods. This can be handled by `define-keyword-syntax-class`.

12 Adding function calls

Up until now, the language extensions we have built have been minor additions to the languages, mostly for convenience. To start building towards a much higher-level language, we build a language with simple function calls. We can additionally augment the type system. We begin by adding a simple facility that permits us to marshal and transmit values across one-way control transfers. Much more sophisticated systems have been designed for

$g \in$ Segment ::= ...
| `(fundef (`f ...`)` s`)`
$a \in$ FunArg ::= n
$f \in$ FunDef ::= `(`a `(`a ...`)` s`)`
$s \in$ Stm ::= ...
| `(funcall` a a ...`)`
| `(arg2reg` r a`)`
| `(exp2arg` a e`)`

Fig. 17. Extending the assembler syntax to add one-way function calls

describing function-call protocols (Olinsky *et al.*, 2006); the system we develop here could certainly be extended and made more powerful.

At this new language level, we define a "function" to be a special label defined at the top level. Functions are defined by using the `fundef` form, and called using the `funcall` form. Functions are called with an arbitrary number of arguments, and never return. Functions that return will be introduced at a higher language level. If we assume a new syntactic form `putchar` that calls a system routine to print a character, we can write code to print the characters `ab`:

```
(fundef ((foo (x y) (seq (arg2reg z x)
                         (putchar z)
                         (funcall y)))
         (bar ()    (putchar #\b)))
  (seq (exp2arg char1 #\a)
       (funcall foo char1 bar))
```

Arguments, both formal and actual, are a type of syntax node separate from ordinary registers. This is mostly convenient for semantic analysis: keeping registers that are only used in function calls separate from those that can be arbitrarily manipulated will help us in later analyses, particularly in calculating control-flow. In order to use arguments outside of the realm of a function call, they must be accessed by using the `arg2reg` and `exp2arg` forms. The `arg2reg` transfers an argument into a register, while `exp2arg` performs the reverse transfer. These forms translate into register moves, which will either be eliminated during register allocation, or will lessen register pressure. The grammar of this new language is presented in Figure 17.

Function arguments are implemented as assembly registers and labels. Their implementation takes the form of a lazy-delegation object that delegates to an expression.

```
(define-class funarg (name register?)
  (λ () (if register?
            (object asm-register name)
            (object asm-label name))))
```

Likewise, `arg2reg` and `exp2arg` translate into `mv` instructions.

```
(define-class arg2reg (dst src)
  (λ () (object asm-mv dst src)))

(define-class exp2arg (dst src)
  (λ () (object asm-mv dst src)))
```

The Ziggurat definition of the function form is fairly simple: for each function call, we allocate a heap frame containing the arguments to the function. At a function call, we allocate the frame and store the actual arguments, and at the function site, we unload the arguments from the arguments frame. While it is not practical to define these as high-level macros, the use of templates simplifies it quite a bit.

```
(define-class fundef (label formals body)
   (λ ()
     (let loop ((formals formals) (num 0))
       (if (pair? formals)
           (parse-template-with-variables
            top-level-asm-env source
            (('fnext (loop (cdr formals) (+ num 1)))
             ('temp-reg (generate-arg-register-object))
             ('formal-reg (car formals)))
            `(seq (add temp-reg arg-reg ,num)
                  (ld formal-reg temp-reg)
                  fnext))
           body))))
```

This code specifically captures an external `arg-reg`; we do this so that we have one register common to all function calls and returns that contains the location of the current argument frame.

12.1 Types for continuation-passing style

In order to add types to our functions, we must alter the syntax a little bit. We now put types on the formal parameters of our functions: $f \in \text{FunDef} ::= (a\ ((a\ t)\ \ldots)\ s)$ Next, we must define a function type:

$$t \in \text{Type} ::= \ldots \mid (t_1 \ldots \rightarrow \bot)$$

```
(define-class function-type (args))
```

We could have our function type delegate to a code type, but we do not, for two reasons:

- We do not wish to bake our function-call mechanism into the type system, as would be necessary to do inside the delegation function.
- It is not necessary, since we wish to enforce that functions are *only* called in the context of a function-call instruction.

Since argument registers are separate from ordinary registers, we must augment the definition of code types to be not a Reg $\rightarrow$ Type map, but rather a Reg $\cup$ FunArg $\rightarrow$ Type map. Now, we can easily specify the new type rules, in Figure 18.

$$\frac{\begin{array}{c}E \vdash a : t \text{ in } ct \\ E \vdash r : t \text{ in inst}(E[\texttt{next}]) \\ \text{inst}(E[\texttt{next}])/r = ct/r\end{array}}{E \vdash [\![\texttt{(arg2reg } r \ a\texttt{)}]\!] : ct}\ \textsc{Arg2RegType}$$

$$\frac{\begin{array}{c}E \vdash e : t \text{ in } ct \\ E \vdash a : t \text{ in inst}(E[\texttt{next}]) \\ \text{inst}(E[\texttt{next}])/a = ct/a\end{array}}{E \vdash [\![\texttt{(exp2arg } a \ e\texttt{)}]\!] : ct}\ \textsc{Exp2ArgType}$$

$$\frac{\begin{array}{c}E \vdash e : (t_1 \ldots \to \bot) \text{ in } ct \\ E \vdash e_1 : t_1 \text{ in } ct \ldots\end{array}}{E \vdash [\![\texttt{(funcall } e \ e_1 \ \ldots\texttt{)}]\!] : ct}\ \textsc{FuncallType}$$

$$\frac{\begin{array}{c}F \vdash s_1 : (a_{11}{:}t_{11}; \ldots) \ldots \\ F \vdash s_b : ct_b\end{array}}{[\![\texttt{(fundef ((}l_1 \ \texttt{((}a_{11} \ t_{11}\texttt{)} \ \ldots\texttt{)} \ s_1\texttt{)} \ \ldots\texttt{)} \ s_b\texttt{)}]\!]\bot}\ \textsc{FundefType}$$

where

$$F = [l_1 \mapsto (t_{11} \ldots \to \bot)] \ldots E$$

Fig. 18. Adding types for continuation-passing style functions

Encoding these is once again simply a matter of encoding the type rules.

```
(define-method funcall make-stm-type-goal
  (λ (s ct e)
    (let loop ((args actuals) (arg-types '()))
      (if (pair? args)
          (fresh (new-arg-type)
                 (make-exp-type-goal (car args)
                                     new-arg-type e ct)
                 (loop (cdr args)
                       (cons new-arg-type arg-types)))
          (make-exp-type-goal fun (object function-type
                                          (reverse arg-types))
                              e t)))))
```

13 Control-flow analysis

In addition to providing a nice link to the type system, the function-call abstraction allows us to do some more complicated analysis. Something we will need later on for compilation, as well as for simple debugging, is a notion of control-flow: we wish to answer the question, "from a particular instruction, where might control go next?" Naturally, we want a conservative analysis: while we can tolerate control never going to a position we thought

it might, we may have problems if control does go to a position we never anticipated. Thus, our low-level analysis might be somewhat broad, while we use the extensibility of Ziggurat to make it more precise when we extend the syntax.

13.1 Fixed-point analysis with Tsuriai

In order to perform global control-flow analysis, we employ a constraint solver called *Tsuriai,* which we developed specifically for Ziggurat, designed much in the same vein as Kanren. Unlike Kanren, however, goals are based on set inclusion rather than unification, and are run until a fixed point is reached.

The basic goal in Tsuriai is an assignment of a set to a variable. For example, if we have two variables x and y, where x is defined by $x = y \cup \{3\}$, and y is defined by $y = x \cup \{4\}$, and we wish to find the least fixed point of these mutually recursive definitions, we would write

```
(fresh (x y)
  (<- x (U y (set 3)))
  (<- y (U x (set 4))))
```

Solving the goal with a `run` command tells us that $x = y = \{3, 4\}$.

We need one more thing in order to start building control-flow analyses. The `fresh` keyword will not suffice in order to introduce logic variables; we will need a logic variable for each statement, since we will need a mapping from each statement to a set of statements. Therefore, Tsuriai provides a function `!` that is a simple mapping from lazy delegation objects to variables.

Now we can define Tsuriai goals for control-flow problems. These are created by the generic function `(make-stm-flow-goal` *s e*`)`, where *s* is the statement in question and *e* is the label environment. Most of these will be quite simple. For example, since a `mv` statement always branches to the statement labeled `next`, its associated goal is:

```
(define-method asm-mv make-stm-flow-goal
  (λ (s e)
    (<- (! s) (label-flow-lookup e 'next))))
```

The methods for `add`, `ld` and `st` are analogous. Flow analysis for `let` and `letrec` simply extends the label environment and calls `make-stm-flow-goal` recursively.

```
(define-method asm-let make-stm-flow-goal
  (λ (s e)
    (let ((new-env (let loop ((labels labels) (stms stms))
                     (if (pair? labels)
                         (extend-label-flow-environment
                          (loop (cdr labels) (cdr stms))
                          (car labels)
                          (car stms))
                         e))))
      (fold (λ (s g) (all (make-stm-flow-goal s e)
                          g))
            (make-stm-flow-goal stm new-env)
            stms))))
```

However, `jmp` and `bez` do present a challenge—in fact, the main challenge—of control-flow analysis: what if we branch to a register? At the base assembly-language level, we give the most conservative answer possible, and use a special value, $\bot$, which in this context is our "don't-know" value: that is, if the set of statements that a statement may go to is $\bot$, then control may go anywhere.[2] The rule for `jmp` is thus dependent on the kind of expression of the destination expression. So, we call a generic function on the relevant expression.

```
(define-method asm-jmp make-stm-flow-goal
  (λ (s e) (make-exp-flow-goal dst s e)))
```

The generic function `make-exp-flow-goal` returns a goal corresponding to the statement `s` branching to the proper target. For example, the `make-exp-flow-goal` on labels is

```
(define-method asm-label make-exp-flow-goal
  (λ (ex s e)
    (<- (! s) (U (! s) (label-flow-lookup e name)))))
```

The generic function `make-exp-flow-goal` on registers then simply gives us $\bot$.

```
(define-method asm-register make-exp-flow-goal
  (λ (ex s e) (<- (! s) bottom)))
```

Using this, we can analyse

```
(seq (add x x 1)
     (jmp x))
```

and discover that the only next statement for `(add x x 1)` is `(jmp x)`, and the set of possible statements for `(jmp x)` is $\bot$.

[2] In actual practice, control cannot go absolutely anywhere; it can only go to labels that have been stored into registers or memory. Therefore, we maintain a flow variable `bottom`, and every time we store a label `l`, we add a goal that `bottom` may contain `l`.

13.2 Control-flow analysis for functions

The fact that we give up on control-flow analysis whenever a statement jumps through a register greatly restricts the kinds of programs we can effectively analyse. For example, the function calls presented above will be completely unanalysable, since every function call involves a jump to a register. However, control-flow analyses for higher-order languages do exist, and we would very much like to take advantage of them.

Using Tsuriai, we can implement a constraint-based version of 0CFA (Shivers, 1991), a basic control-flow algorithm for higher-order languages. But first, we'll need to add an additional function to Tsuriai, `maps`. The function `maps` takes an accessor and a function from one argument to a Tsuriai goal. This function then returns a Tsuriai goal. Running `maps` on an accessor a and a function f applies f to each value returned by a. So, in order to compute the set S defined by

$$1 \in S$$
$$\forall n \in S \,.\, 3 - n \in S$$

we write the Tsuriai goal:

```
(fresh (S)
  (<- S (U S (set 1)))
  (maps S (λ (n) (<- S (U S (- 3 n))))))
```

Tsuriai does not give us the computational power we would need for true abstract interpretation, but we can run a constraint-based algorithm. We use Tsuriai to find a set of possible values for each funarg. At this language level, control-flow is manipulated through higher-order functions, and our algorithm is explicitly designed to take this into account: we only consider mappings of funargs to sets of fundefs; all other possible assignments, for example, through the use of `exp2arg`, cause the funarg to be mapped to a conservative $\perp$ value.

The algorithm is implemented as the generic function `make-stm-flow-goal` for fundefs and funcalls. For fundefs, all we need to do is map the function name to its definition.

```
(define-method fundef make-stm-flow-goal
  (λ (s e)
    (all (<- (! (funarg-name label))
             (U (set s) (! (funarg-name label))))
      (make-stm-flow-goal body e))))
```

The reason we have the goal

```
(<- (! (funarg-name label))
    (U (set s) (! (funarg-name label))))
```

instead of the apparently equivalent `(<- (! (funarg-name label)) (set s))` is that the former allows for the possibility that the funarg that represents the current function might be overwritten with an `exp2arg` instruction. Now, for each function call, we map the formal arguments to the actual arguments.

$g \in$ Seg ::= ...
| `(function-language` s`)`
$s \in$ Stmt ::= ...
| `(let-fun (`f ...`)` s`)`

Fig. 19. Extending the assembler syntax to provide lexically-scoped function declarations

```
(define-method funcall make-stm-flow-goal
  (λ (s e)
    (maps (! (funarg-name fun))
          (λ (fundef)
            (map-formals-to-actuals fundef actuals)))))

(define-method fundef map-formals-to-actuals
  (λ (fundef actuals)
     (let loop ((formals formals) (actuals actuals))
       (if (and (pair? formals) (pair? actuals))
           (all (<- (! (funarg-name (car formals)))
                    (! (funarg-name (car actuals))))
                (loop (cdr formals) (cdr actuals)))
           succeed))))
```

The rules for `exp2arg` returns $\perp$, which is pre-assigned to the set of all functions.[3] The `succeed` goal succeeds automatically, without binding any variables.

14 Closure conversion

While now we have functions, we are still missing many of the features of a high-level programming language. In the next step, we attempt to start using lexically-scoped variables as opposed to the registers available lower than this point. In order to make this conversion, we have to start using closures.

14.1 Function flattening

At the simple function-call language level, all of our functions are introduced at the top level. At higher language levels, we will need the ability to introduce functions in the middle of the program. So, we introduce a `let-fun` form, as in Figure 19.

The `function-language` form indicates that the program may contain local functions. It acts as a wrapper, allowing us to embed source written in the language that provides lexically-scoped function declarations into the language context of the lower-level language, which only permits top-level, globally-scoped function declarations. The wrapper is, of course, a macro that compiles its body into a `fundef` form.

[3] Once again, in actuality, $\perp$ need not map to every function, merely the ones that escape via `arg2reg`.

```
(define-class cl-function-language (stm)
   (λ () (syntax-object fundefs source
           (local-fundefs stm)
           start)))
```

The function definitions of the rewritten `fundef` form are discovered as a result of invoking the `local-fundefs` generic function on the enclosed statement. The generic function `local-fundefs` returns all of the function definitions made inside an assembly statement. We need to define this method for all assembly-language statements. For `let` and `letrec` statements, it will combine the local function definitions of all of the statements inside that statement. For all other base-assembler forms, it will return the empty set.

```
(define-generic (local-fundefs stm) '()))

(define-method asm-let local-fundefs
  (λ (this)
    (append (local-fundefs stm)
            (foldl append '()
                   (map local-fundefs stms)))))
```

Local `let-fun` forms will return the functions defined.

```
(define-method let-fun local-fundefs
  (λ (this) (append fundefs
                    (foldl append '()
                           (map local-fundefs fundefs))
                    (local-fundefs start))))
```

Since function definitions are lifted to the top level, the `let-fun` form, when it is compiled, simply removes the function definitions.

```
(define-class let-fun (fundefs start)
  (λ () start))
```

This will allow us to define functions locally.

14.2 Defining closures

The argument registers introduced in the previous section have different scoping rules from ordinary registers. They are declared in the function header, and are accessible within the scope of the function definition. With locally defined functions, this presents us with a potential problem: argument registers defined in one function may be used in another. Therefore, we need closures.

The purpose of a closure is to store the values bound to the free variables in a function, thereby "closing" it. Therefore, in order to create closures, we need to know the free variables contained in a function. We do this via a lazy-delegation method, `free-variables`, that, when called on an AST node, returns an object of class `arg-set`. An `arg-set` represents an unordered collection of funargs, and has the usual set operations, `arg-set-union`, `arg-set-remove`, and so forth, defined as generic functions.

For the base assembler types, the implementation of `free-variables` is trivial. For `asm-let` and `asm-letrec`, `free-variables` merely calls itself recursively on the subforms.

```
(define-method asm-let free-variables
  (λ (this)
    (arg-set-union
     (foldl arg-set-union (object arg-set '())
            (map free-variables stms))
     (free-variables stm))))
```

For all other base forms, `free-variables` merely returns the empty set.

```
(define-method asm-jmp free-variables
  (λ (this) (object arg-set '())))
```

It's up to the higher language level to actually generate free variables. Both `funcall` and `arg2reg` introduce free variables.

```
(define-method arg2reg free-variables
  (λ (this) (object arg-set `(,src))))

(define-method funcall free-variables
  (λ (this)
    (arg-set-union (object arg-set `(,funref))
                   (foldl arg-set-union (object arg-set '())
                          (map (λ (x) (object arg-set `(,x)))
                               actuals)))))
```

Function definitions can bind variables, and thus remove free variables from a statement.

```
(define-method fundef free-variables
  (λ (this)
    (arg-set-subtract (free-variables stm)
                      (object arg-set formals))))
```

Mutually recursive function definitions via `let-fun` are a little more complicated. Since such a definition binds a set of funargs to function definitions, it removes those funargs from the set of free variables inside those functions.

```
(define-method let-fun free-variables
  (λ (this)
    (arg-set-subtract (arg-set-union
                       (free-variables stm)
                       (foldl arg-set-union (arg-set '())
                              (map free-variables fundefs)))
                      (object arg-set
                              (map fundef-arg-reg fundefs)))))
```

Now that we know the free variables found in a function, we can build closures. At the closure-language level, references to functions as values are closures, rather than simple code pointers. A closure is a block in memory, containing first the pointer to the function code, followed by the value of each of the free variables of that function. In addition, all functions will take an additional argument, the closure of the function being called. Because of that, we will need new definitions for `let-fun`, `funcall` and `fundef`.

At this language level, `funcall` merely extracts the code pointer and calls it, with the additional argument.

```
(define-class cl-funcall (funref actuals)
  (λ ()
    (let ((temp-reg (generate-temp-register-object)))
      (parse-template-with-variables top-level-asm-env source
        (('temp-reg temp-reg)
         ('funref funref)
         ('fcall (object funcall source temp-reg
                         (cons funref actuals))))
        `(seq (ld temp-reg funref)
              fcall)))))
```

The job of a closure is to bind all of the free variables of a function. Thus, when a function is called, the first thing it does is to take all of the values from the closure and put them back into their respective values.

```
(define-class cl-fundef (name label first-formal
                                 formals body)
  (λ ()
    (object fundef source label
            (cons first-formal formals)
      (let loop ((closure-vars
                  (arg-set->list
                   (arg-set-subtract
                    (free-variables body)
                    (object register-set formals))))
                 (num 1))
        (if (pair? closure-vars)
            (parse-template-with-variables
              top-level-asm-env source
              (('fnext (loop (cdr closure-vars) (+ num 1)))
               ('free-reg (car closure-vars))
               ('first-formal first-formal))
              `(seq (add temp-reg first-formal ,num)
                    (ld free-reg temp-reg)
                    fnext))
            body)))))
```

Closures are generated when the function is declared. Closures are blocks of memory that contain the free variables of a function.

```
(define-method cl-fundef cl-fundef-make-closure
  (λ (this)
    (let ((interior
           (let loop ((closure-vars (arg-fun-set->list
                                     (free-variables this)))
                      (num 1))
             (if (pair? closure-vars)
                 (parse-template-with-variables
                    top-level-asm-env source
                    (('fnext (loop (cdr closure-vars)
                                   (+ num 1)))
                     ('free-reg (car closure-vars))
                     ('reg name))
                    `(seq (add temp-reg reg ,num)
                          (st temp-reg free-reg)
                          fnext))
                 (parse-template-with-variables
                   top-level-asm-env source ()
                   '(jmp (star next)))))))
          (closure-size (+ 1
                           (length (register-set->list
                                    (free-variables body)))))
      (parse-template-with-variables top-level-asm-env source
        (('interior interior)
         ('register name)
         ('label label))
        `(seq (malloc register ,closure-size)
              (st name label)
              interior)))))

(define-class cl-let-fun (fundefs stm)
  (λ ()
    (object let-fun fundefs
            (let loop ((fundefs fundefs))
              (if (pair? fundefs)
                  (parse-template-with-variables
                   top-level-asm-env source
                   (('fnext (loop (cdr fundefs)))
                    ('build-closure
                     (cl-fundef-make-closure (car fundefs))))
                   `(seq build-closure fnext))
                  stm)))))
```

```
g ∈ Segment ::= (cps c)
c ∈ CPS     ::= (cont s c)
              | (let-fun (f ...) c)
              | (app a (a a ...) c)
              | (if a c c)
              | (return a)
f ∈ FunDef  ::= (a (a ...) c)
```

Fig. 20. Extending the assembler syntax to provide CPS functions

One side-effect of this structure is that, since we use funargs that are available at lower language levels, we do not need to alter our control-flow algorithm at all to incorporate this new functionality.

15 Full function calls via CPS conversion

Up until now, the functions we use never return. In order to have a true conventional high-level language, we need to rectify this. We do this by using our one-way functions to represent continuations, which package up the computation that needs to take place after a function call completes as a functional value that may be passed as an extra argument to the call.

In order to do this, we can not use the control primitives presented above, but we can use basic assembly language. Our function language thus resembles a function-call wrapper on top of assembly language, as seen in Figure 20.

The CPS language has a small set of primitives that perform all of the control manipulation we need.

- `let-fun` defines one or more functions, similar to `let-fun` in the closure language. Similarly to the closure language, we use funargs as arguments, not ordinary registers.
- `cont` sequences instructions. The CPS form `(cont` s c`)` executes the statement s followed by the CPS expression c. This is the only way to use an assembly-language instruction directly in the CPS language.
- `app` calls a function. The form `(app` a_d `(`a_f a_1 `...)` c`)` calls the function a_f with the arguments $a_1, \ldots$, and puts the result in a_d before continuing to c.
- `return` returns from a function. The form `(return` a`)` returns the value a from the current function.
- `if` performs branching. The form `(if` a c_t c_f`)` evaluates a; if its value is 0, control moves to c_t, otherwise, to c_f.

These control primitives can be directly expressed in the closure language. The major change that we make is that we add an argument to every function, to represent the return continuation.

```
(define-class cps-cont (stm cont)
 (λ ()
   (parse-template-with-variables top-level-asm-env
    `((stm ,stm)
      (cont ,cont))
    (let ((*next cont)) stm))))

(define-class cps-app (dst fun args cont)
  (λ ()
    (parse-template-with-variables top-level-asm-env
     `((dst ,dst)
       (fun ,fun)
       (args ,args)
       (cont ,cont)
       (cont-var ,(generate-temp-register)))
     (let-fun ((cont-var (dst) cont))
       (funcall fun cont-var . args)))))

(define-class cps-fundef (name args exp return-funarg)
  (λ () (object cl-fundef name (cons return-funarg args) stm)))

(define-class cps-let-fun (fundefs stm)
  (λ () (object cl-let-fun fundefs stm)))

(define-class cps-return (val return-funarg)
  (λ () (object cl-funcall return-funarg val)))
```

Note that `cps-fundef` and `cps-return` have an extra field: `return-funarg`. This field contains the funarg with the current return continuation. We determine this when the CPS expression is parsed; therefore, we need a new sort of static environment, to contain the current `return-funarg`.

```
(define-class return-funarg-env (return-funarg super)
  (λ () super))

(define-generic (current-return-funarg env))

(define-method return-funarg-env current-return-funarg
  (λ (this) return-funarg))
```

The parsing functions thus need a minor change, in order to incorporate the current `return-funarg`. The function definition needs to build a new `return-funarg-env` in order to introduce a new `return-funarg`.

```
c ∈ CPS         ::= ...
                  | (let-syntax ((n t) ...) c)
g ∈ Seg         ::= ...
                  | (define-syntax n t)
t ∈ Transformer ::= (syntax-rules (n ...) ((p c) ...))
p ∈ Pattern     ::= (n (n u) ... [...])     ; Optional "..." is literal keyword.
u ∈ Parser      ::= c
```

Fig. 21. Extending the assembler syntax to provided CPS macros

```
(define (parse-cps-fundef env form)
 (let ((return-funarg (object funarg (gensym)))) ; fresh var
   ((parse-pattern-variables
     `(,parse-funarg ,(parse-list-pattern parse-funarg)
                     parse-cps-exp)
     (name args exp)
     (object cps-fundef name args exp return-funarg))
    (object return-funarg-env return-funarg env) form)))

(define parse-cps-return
 (parse-pattern-variables env
  `(return ,parse-funarg)
  (_ val)
  (object cps-return val (current-return-funarg env))))
```

Thus, we have built a compiler for functions out of macro primitives.

16 Macros on a higher-level language

Now that we have closures and continuations, we have something that at least resembles a higher-level language. But just because we've reached the top of the tower is no excuse to stop building! We can very easily put a new macro system on our language, and permit a whole new breed of semantic conveniences. But what do we do with these macros? What do they look like?

We can add high-level macros with the same facility as at lower-level languages. We start by creating a new control primitive.

```
(define-syntax seq
  (syntax-rules ()
   ((seq (x cps-exp)) x)
   ((seq (x asm-stm) (y asm-stm) ...)
    (cont x (seq y ...)))))
```

We should note here that we have two definitions of `seq`: one for an assembler context, and one for a CPS context. It is perfectly allowable to use them both at the same time; since we have different parsers for different syntax types, we will parse the keyword `seq` as an

assembler statement when in an assembly context, and a CPS expression when in a CPS context.

Now that we have higher-order functions, we can build more advanced macros, such as macros that take functions as arguments.

```
(define-syntax box-function
  (syntax-rules ()
   ((box-function (f funarg))
    (seq (arg2reg v f)
         (malloc mem 1)
         (st mem v)
         (exp2arg ret v)
         (return ret)))))
```

This takes a funarg, allocates it in its own cell in memory, and returns the location of that cell.

17 Variations on the design of Ziggurat

It's worthwhile to consider variations on the specifics of Ziggurat's design. Two such variations are (1) extending the system to work with more general grammars, and (2) providing the Ziggurat meta-language with a static type system to guarantee Ziggurat code constructs well-formed source terms.

17.1 From S-expressions to general grammars

The current Ziggurat technology commits to representing all languages within the general framework of S-expressions. This is a reasonably universal framework: any context-free grammar (CFG) can be mapped to an S-expression grammar in a straightforward way.[4] It's possible to think of S-expression grammars as the ultimate LL(1) "predictive" syntax: a form is introduced by left-parenthesis/keyword. Once we've seen this prefix, we know what production of our grammar has been selected. Predictive, or LL, grammars have the pleasant property that they are easy for humans to read in sequential, left-to-right order: the reader never has to read and "buffer up" an arbitrary amount of text before being able to determine its grammatical structure.

However, while S-expression notations may have universal scope, they are not universally adored. It's reasonable to wonder if we could adapt Ziggurat to work with notations based on more general context-free grammars, *e.g.*, LL(k) or LALR(k) grammars. If we were to attempt to do so, there would be a new set of technical challenges to be overcome, distinct from the ones we have addressed in this article.

First, note that LL(k) and LR(k) grammars do not compose. Suppose, for example, we have an LALR(1) grammar for Java, and we design a second LALR(1) grammar for regular expressions. If we mix these two grammars together to produce an extended Java that has

[4] To see this, simply assign each production in the original grammar a unique keyword for its S-expression analog.

a regexp form, the union of the two grammars is not necessarily LALR(1). Suppose we abandon modularity and tune our regexp grammar so that, when added to the core Java grammar, the result remains LALR(1). Now suppose that some other programmer does the same for an SQL grammar, intended to extend Java for database queries. Even given that these two language extensions are LALR(1), we still have no guarantee that the union of the two will preserve LALR(1).

Second, recall that a lexically-scoped macro facility requires parsing and static analysis to be intertwined (as discussed in Section 4.3). An attractive feature of S-expressions is that they are a form of "pre-grammar" (in the same sense that the XML community refers to XML terms as "semi-structured" data). S-expressions have a limited amount of structure—the tree structure of the sexp—that has an important interaction with our interleaved parsing/analysis task. Consider, for example, what happens when a Scheme compiler encounters a form such as

```
(q (a b) 3 "dog")
```

Even before the compiler has determined what the form *is* (*e.g.*, is it a macro? a function call? a core-language special form?), it is able to *bound* the form's syntactic extent: it ends at the right parenthesis that matches the "`(q`" introduction. So, if `q` is a macro keyword, the compiler knows exactly how much source text (or, more accurately, how much of the pre-grammar tree) to pass to its transformer: the material within the parentheses.

One possible avenue one could explore in order to move beyond S-expression syntax is to employ parsers capable of parsing general context-free grammars, such as Packrat (Ford, 2002) or other GLR techniques. We lose the ability to parse input with linear time and memory, but this is not of critical importance for many applications. While general CFGs compose, of course, this does not resolve the issue of ambiguous parses, so this avenue does not provide the kind of modularity we really want.

The ambiguity issue of GLR parsing carries with it a design message. GLR parsers work, roughly, by carrying forward multiple parses in a nondeterministic fashion; hopefully, later information encountered during the parse will winnow the multiple possibilities down to a single unambiguous parse. The fact that GLR parsing requires super-linear computational resources is a sign that we are asking too much of the parser—in particular, when the parsing engine happens to be a human. Notations should be easy to read. Asking a human to hunt around a large string playing detective to resolve local syntactic ambiguity is forcing the human to burn time and effort on the wrong task. One is jerked from the semantic level of processing the text up to shallow issues of syntax; the interruption of one's train of thought is annoying and counterproductive. This is bad design.[5]

[5] An illuminating design case is the syntax of APL. Because the language has so many primitive operators, one would expect APL's precedence rules to be a nightmare—significantly worse than the rules for C's modest set of operators (which are already too complex for most programmers to remember and use correctly). However, APL's solution to this Gordian knot is elegant and simple: APL resolves operator application right to left. At first, it seems odd that "`a × b + c`" means "add `b` and `c`, then multiply this sum times `a`," but after a week of programming, it becomes a great relief that the programmer never has to interrupt his train of thought to resolve operator precedence: a single universal rule handles every situation.
Note that S-expression frameworks, like APL syntax, have no precedence issues, for a similar reason: they are too simple to have problems. Scheme programmers never, ever spend time wondering about the precedence of operations in the string "`(* a (+ b c))`."

A more promising approach is to *restrict* our grammars instead of enriching them, relative to LR(k). The "parsing expression grammars" of Grimm's Rats! system (Grimm, 2006) for syntax extension *are* truly composable. It might be possible to attach a Rats!-like parser-generator to a Ziggurat-like macro system, such that language extensions consist of additions to the concrete syntax of the language, to form a composite syntax for each extension. Coming from a Scheme-macro background, however, we still would like a technology that permits syntax, itself, to be bound with lexical scope, as described above.

Again, could one figure out some way to beat these problems in a setting that permits more general grammars? Perhaps. However, we do not believe that this is the deep problem with the large payoff. The expressiveness gained when moving from an S-expression framework to a general CFG framework is, to our way of thinking, simply a "constant factor" in notational overhead. An aphorism of the Lisp/Scheme community states that after a few weeks of programming in these languages, "the parentheses disappear." That is, programmers no longer perceive the program in terms of the pre-grammar; they perceive, rather, the actual syntactic structure of the terms, which is as rich in the S-expression world as it is in the general CFG world. The true payoff comes in having a general framework in which we can express different notations with distinct static and dynamic semantics. This is the theme we have explored with our work here.

Scheme programmers can, and do, routinely write code where they shift between multiple languages in the space of a few lines of code. In the space of ten lines of code, a Scheme program can shift between core Scheme syntax, specialised process notation for executing Unix processes, regular expressions for searching strings, and an HTML notation for scripting web interactions. The syntactic cost to shifting between these languages is nothing more than a left parenthesis and a single keyword; the specialised-language term extends to the matching right parenthesis. Until we can see how one could achieve this kind of syntactic modularity and composability in a more general CFG framework, we will have a difficult time working up much interest in this larger setting. As programmers, we are being too richly rewarded for staying within the S-expression paradigm.

17.2 Static semantics in Ziggurat

Ziggurat provides no form of static constraint (at the Ziggurat, or meta, level) to guarantee that source terms constructed by Ziggurat code are even syntactically well-formed trees. Ziggurat code is able to build a syntax tree that has, *e.g.*, a "declaration" or "statement" node where an "expression" node should go. It is up to the designer of a given language to implement code that walks syntax trees to check them for correctness. Could we ex-

A second design heuristic that is related here is the notational mantra, "Always use a tool small enough for the job." We believe that the *clarity* of a notation (to a human) and the degree to which terms in that notation can be *analysed* (by a computer algorithm) are related: if a term is difficult for a computer to understand, it is quite likely going to be difficult for a human to understand. Thus, expressing a computation with a regular expression (when we can do so) is likely clearer than expressing that computation in C code.
This applies to parsing, as well: *ceteris paribus*, it seems reasonable to suppose that a notation that can be expressed with an LL(1) grammar, which can be parsed with a linear-time/linear-space algorithm, is going to be easier for a human to read and parse than a notation whose grammar requires a super-linear parsing algorithm.
Notations are designed artifacts; there is benefit to lightweight designs.

tend Ziggurat with, say, a static type system that would guarantee that the syntax trees it constructs are correct?

We believe it would be possible to do so, and that such a type system would provide a useful capability to language designers working in Ziggurat. The recent work of Herman and Wand (2008), for example, is one promising approach towards addressing this issue. We did not explore such a type system simply in order to simplify the task of designing and implementing Ziggurat; it would have delayed or halted development and evaluation of the system we currently have.

Note, also, that syntactic correctness is just the beginning. Simply guaranteeing that a term is *grammatically* well-formed (which is what a type system would provide, at the Ziggurat, abstract-syntax meta-level) does not guarantee that the term will pass more sophisticated tests associated with its language's static semantics. So having a type system in the meta language does not save us from the necessity of doing static correctness checks with Ziggurat code.

18 Related work

The subject of language extension has recently been of interest to a number of projects, due in part to the proliferation of domain-specific languages for Web services. Ziggurat's design has been influenced by several of these related projects.

18.1 Metaprogramming

Some recent work in domain-specific macros has been in "metaprogramming," the development of tools related to source-to-source program translation. A successful metaprogramming tool in wide use today is Stratego (Visser, 2004), a program-to-program source code translation toolkit based on term-rewriting systems.

Stratego allows one to design a domain-specific language, and specify its dynamic semantics by rewriting programs into an underlying language, such as Java. Thus, Stratego makes it easy to write source-to-source compilers, which is similar to the functionality presented by macro systems. An advantage of Stratego over Ziggurat and other Scheme-like macro systems is that Stratego allows for non S-expression-based grammars. However, Stratego does not provide tools for semantic extension. In order to implement a type system for a domain-specific language designed with Stratego, the source-to-source compiler would first check the types of the source program, translate the program text, and then allow the compiler of the underlying language to perform its own type checking. It is not possible to link the static semantics of the domain-specific language to that of the underlying language, as in Ziggurat. This complete separation of language semantics is good for translating languages that have dissimilar semantics, such as translating from a database-query language to a high-level language, but does not work as well for finer-grain language extension, such as the `seq` and `kons` language extensions presented earlier in this paper.

Another difference between Stratego and Ziggurat is that Stratego is a tool for full-program transformation, and does not allow languages to be embedded, as in macro systems. However, the MetaBorg tool (Bravenboer & Visser, 2004), which is built on top of Stratego, does allow languages to be embedded. For example, one could implement a

regular-expression language or a GUI-specification language inside of Java in MetaBorg—tasks one might accomplish with a macro system. However, language embedding is still coarser-grained than syntactic extensions such as `seq` and `kons`, which are possible with macro systems. In MetaBorg, it is not possible for source-language terms to appear in the same syntactic context as embedded terms. Ziggurat aims to augment the syntactic extensions provided by macro systems with corresponding semantic extensions, whereas MetaBorg aims to allow languages with dissimilar static semantics to be embedded.

A system with a similar approach to Stratego's is Metafront (Brabrand *et al.*, 2003), a tool for specifying languages with the goal of making them extensible. Metafront allows a language extension to specify how it adds to the grammar of a language, and then to provide rules to transform programs in the extended language to the base language. Metafront, like Stratego, is a full-program transformation toolkit, without semantic extensibility. In addition, the transformation rules are not Turing-complete, which guarantees that the compiler will terminate, but does not allow the power of full low-level macros.

An alternate approach to metaprogramming is multi-stage programming, implemented in MetaML (Taha & Sheard, 1997). MetaML takes many ideas from macro technology. In MetaML, syntax is manipulatable data, similar to the way macros translate syntax. Programs thus can output programs, which, in turn, can output other programs, providing staged compilation similar to macros. Semantic analysis is possible through a form of reflection: variables defined in one compilation stage are available in later stages. However, as in Stratego, the focus is on full-program transformation, unlike the local syntactic extensions available with macro systems.

Nanavati (2000) also provides a system for extensible analysis. However, Nanavati does not provide an object-oriented interface to his syntax. In order to implement significant syntactic extensions in his system, the programmer is forced essentially to simulate an object-oriented architecture programmatically, which makes for a fairly awkward, tortured coding style. Ziggurat captures this structure directly in the linguistic mechanisms of the meta-language.

18.2 Macro systems

The implementation of syntax as objects, first introduced by Dybvig, Hieb and Bruggeman (Dybvig *et al.*, 1992), opens up numerous possibilities.

The analysis of macros themselves is an important related topic. Programs containing macros can complicate tasks that are simple in languages that do not contain them. For example, macros complicate separate compilation: if one module depends on another module, it may depend on macros found within that module, meaning that compilation of the first module will require including the source of the second. Flatt (2002) has developed a method of mixing the specification of macros with the specification of modules, allowing modules to be partially imported at compile-time, as necessary for macros, and thus allowing for separate compilation. The analysis of the interaction of macros and modules is an important step in applying macro technology to a language with a module system, one that is not covered by this work.

These compilable macros are implemented in the macro system currently employed in MzScheme. MzScheme uses syntax objects; an earlier version of the macro system, called

McMicMac (Krishnamurthi *et al.*, 1999), performed semantic analyses on syntax. Ziggurat draws many ideas from McMicMac. However, while Ziggurat attempts to be a general-purpose language toolkit, McMicMac is designed exclusively for Scheme, only dealing with the limited static semantics available in Scheme.

Others have mixed hygienic macros with languages with static semantics, without providing an explicit method of accessing those semantics. The language Dylan (Shalit, 1996) contains a hygienic macro facility, without such a mechanism. The Java Syntax Extender (Bachrach & Playford, 2001) adds a hygienic macro mechanism to Java, again with no means to access the static semantics of the language. Even without such a mechanism, these systems are adequate for many sorts of language extensions.

The Maya macro system (Baker & Hsieh, 2002), on the other hand, allows macros to access the types of subexpressions. It does this by performing macro expansion in stages, delaying expansion until the types of subexpressions are available. This is similar to the laziness of lazy delegation, but it does not allow macro writers to specify their own static semantics.

18.3 Attribute-grammar approaches

A popular method of specifying static semantics is via attribute grammars. Another approach to mixing language extension and static semantics is to augment semantics specified via attribute grammars with metaprogramming facilities.

The JastAdd extensible Java compiler (Ekman & Hedin, 2004; Ekman & Hedin, 2007) uses attribute grammars to enable tree rewriting, similar to the Stratego language-extension tool. Like Stratego and other metaprogramming tools, JastAdd is based on rewriting one full language to another, making it ideal for incrementally building a compiler.

Silver AG (van Wyk *et al.*, 2007) is a language for describing programming-language grammars and their associated static semantics with the goal of making these languages extensible. An application of this system is the Java Language Extender framework, a tool for writing extensions to the Java language, both fine-grained, such as new loop constructs, and coarse-grained, such as embedding domain-specific languages.

Silver is based on an attribute-grammar framework. A language is described as a number of syntax productions. Each production defines a number of attributes, compile-time values which are calculated from the attributes of its ancestors in the abstract syntax tree ("inherited" attributes) and its children ("synthesized" attributes). Each non-terminal defines a fixed set of attributes to be calculated for each production of that non-terminal.

In Silver AG, the language extender is only required to provide definitions for a subset of the attributes defined on a production. This is enabled by a feature called forwarding (van Wyk *et al.*, 2002). Productions can have a special attribute, called its forwards-to attribute, which is similar to the delegate in Ziggurat. To calculate the value of any attribute not defined on a production, Silver AG calculates the production's forwards-to attribute, which should return a syntactic node, and then delegates to this node, similar to how lazy delegation deals with generic functions on objects with undefined methods.

The difference between Silver and Ziggurat involves how they deal with the interleaving of parsing and analysis. Both allow this, quite explicitly. In Silver, the forwards-to attribute is an attribute like any other, and thus can be calculated from the attributes of the node's

ancestors and descendants. Since attributes are calculated based on the values of other attributes, the order of evaluation of attributes is very important, and is determined via data dependence; the attribute specification is purely functional, and thus expansion can be delayed until it is required. Thus, if the forwards-to attribute depends on other attribute values, Silver will perform expansion after other semantic analyses.

Ziggurat allows this behavior by means of the laziness in lazy delegation. Ziggurat uses only simple data-flow to order the computations defined in the meta-language: it does not perform expansion until it is needed by a generic function being applied to a syntactic node lacking a method for that function. This puts the burden on the language designer to prevent expansion from taking place until the information necessary for it is available, but allows expansion in cases where it may be difficult to automatically determine the order in which things should be evaluated. We have found this linguistic on-demand mechanism fairly natural and intuitive to use.

Ziggurat's ability to delay expansion using lazy delegation is an important design element of the system. Sophisticated macros that mediate shifts between language levels can be written to exploit static-semantics computations performed at the higher language level to drive the rewrite into the lower language level. For example, the expander (*i.e.*, the delegation method) can perform type-dependent translation, or use the results of a flow analysis to select between translation strategies.

Consider implementing named, static memory locations in our assembly language. We would like to allocate a block of memory at an early point in the program, and then refer to offsets into this block of memory. We would like to write:

```
(with-static-data
  ...
  (ld-static x speed)
  (ld-static y speed-increment)
  (+ z x y)
  (st-static speed z))
```

and have this expand into:

```
(seq
  ...
  (malloc global-table-location 2) ;number of data slots = 2
  (+ tmp global-table-location 1)  ;speed offset = 1
  (ld x tmp)
  (+ tmp global-table-location 2)  ;speed-increment offset = 2
  (ld y tmp)
  (+ z x y)
  (+ tmp global-table-location 1)  ;speed offset = 1
  (st tmp z))
```

Here, `malloc` is a macro that expands into a branch to a system routine. There are two pieces of data that must be calculated in order to allow expansion to take place:

- The expansion of `with-static-data` needs to know how many static locations are

being used by the program, in order to calculate how large a block of memory to allocate.
- Each use of a static location must know what numerical offset into the static block is associated with that variable name.

In Ziggurat, we implement this information as a field in the `with-static-data` node. Each time the parser encounters a `ld-static` or `st-static` node, it finds the offset associated with the variable name. If there is no offset yet defined, it defines it, and adds it to the mapping. If this mapping were implemented as an attribute, it would create a circular dependency: calculating the offsets requires the table, and the table is generated by calculating the offsets. This would be difficult to implement in Silver AG in such a way that it did not result in a set of attributes without a valid ordering.

Forwarding is based on an earlier language-extension tool, called XL (Maddox, 1989). In XL, syntax nodes are parsed into records containing the attributes of that node. New static semantics are defined by extending these records, using a subtype-based extensible record facility built into the language. However, these records are not composable, limiting the macro designer to a linear tower of languages; macro packages cannot be combined in the same program. These extensible records have associated with them a subtyping relation on sets of attributes, similar to the attribute-grammar inheritance systems found in the LISA tool (Mernik *et al.*, 1999).

18.4 Semantic extension

An important related field of research is projects that have looked into semantic extension of languages, independent of macros. The JavaCOP project (Andreae *et al.*, 2006), for example, looks at extending Java's type system with type constraints. However, JavaCOP's syntactic mechanism only allows programmers to annotate their programs, instead of providing new type constructs. An interesting piece of future work for Ziggurat would be to see if lazy delegation could provide better syntax for JavaCOP's type annotations.

The J& project (Nystrom *et al.*, 2006) establishes a language feature called "nested intersection." Not only is nested intersection an interesting language feature to implement via macros, but an application of their work is in providing fine-grain compiler extensibility through method inheritance. It is a possible future direction for Ziggurat research to explore combining nested intersection with lazy delegation to provide better, more terse descriptions of static semantics.

19 Future work

The potential scope of Ziggurat is wide, and there are many possible avenues one could explore going forward. Our focus is currently on the following.

- **Certification.** Ziggurat permits syntax implementors to intercept and override analysis requests with their own methods. This is a key source of power for the system, but it is a two-edged sword: overloaded analyses, if they are buggy or malicious, can give wrong results, possibly leading to unsafe compilation. Since Ziggurat is intended as a general language-extension toolkit, this is unavoidable in the general

case. A possible solution, though, is to require that certain analyses provide a certificate that their result obeys certain properties, *e.g.*, that they give a correct answer (of possibly varying precision), or that they do not violate safety rules. Certification in this scheme becomes just another analysis.

In general, if a lower-level language includes checkably sound annotations expressing the support for a given assertion, then an analysis performed at a higher level in the tower can be expressed by expanding into annotated code in the lower-level language. Since these annotations will be checked by the static-semantics elements of the lower-level language implementation, there is no possibility of a buggy or malicious macro asserting a false claim undetected.

- **Further analyses.** The basic analyses presented here show the feasibility of Ziggurat for implementing important code analyses. We intend to implement more in Ziggurat. For example, we have implemented only a constraint-based version of 0CFA. We intend to implement a CFA based on abstract interpretation, allowing for more precise higher-order flow analyses, such as 1CFA and ΓCFA (Might & Shivers, 2006).
- **Further languages** We do not have enough experience with Ziggurat to understand the limits to its general "language tower" approach to design. A better sense of the technology's strengths and weaknesses will only come with time.

  The most ambitious example we are attempting is a LALR parser generator, whose exported language is a notation for writing down a context-free grammar with associated semantic actions. Parser generators are an interesting application of Ziggurat technology. First, they have a great deal of static semantics that is completely independent of the underlying target language to which the parser is compiled. The vocabulary of grammar analysis—first and follow sets, LALR(k), SLR, and so on—constitute a static semantics, one where the language's identifiers are not variables, but non-terminals, which are defined and referenced by the various productions of the grammar.

  This leads to the second interesting issue: how will it affect the parser-definition task to permit *grammar* macros, that is, macros that occur in production right-hand sides, and expand into terminal/non-terminal sequences?

  Third, the translation between a context-free grammar and the source code that parses strings from that grammar is a sophisticated and complex one: the Deremer-Pennello algorithm (DeRemer & Pennello, 1982) is not simple code. Thus, we push the boundaries of what has traditionally been accomplished via macros—it makes a good example of an arena where there is no hope of statically analysing macro-using code unless the macro itself explicitly provides static semantics, as in Ziggurat.

  Fourth, a parser tool is naturally defined by means of multiple layers in a language tower: a context-free grammar is translated (*e.g.*, by the Deremer-Pennello algorithm) to a specification of a push-down automaton (PDA). Given a language for describing PDAs (*e.g.*, one with "shift," "reduce" and "goto" as basic forms), we can then provide multiple translations from the PDA language to implementing languages: we can compile the PDA directly to control constructs in the lower language (Pennello, 1986), or translate the PDA to a data structure which is then fed to an interpreter (as the tools Yacc and Bison do (Johnson, 1979)); additionally, we can consider performing analysis and source-to-source translations on the PDA program

to improve the performance of the parser. The PDA language has its own interesting static semantics: for example, Pottier has recently shown how important safety invariants on the PDA's stack useage can be statically captured by type systems (Pottier & Régis-Gianas, 2005).

We are basing our Ziggurat implementation on an existing LALR tool we have previously designed and implemented in Scheme (Flatt *et al.*, 2004).

- **Analysis-driven translation.** One of the benefits of lazy delegation is that it allows translation to be delayed until after analysis takes place. Currently, this is used in register allocation, although the framework would allow type-directed translation, for example.

20 Acknowledgements

The reviews we received from our anonymous referees were spectacularly thorough, detailed and thoughtful. The paper was greatly improved by their comments; we are grateful to our editor, Julia Lawall, and the article's referees. This work was funded by the NSF's Science of Design program, and by a grant from Microsoft Research, sponsored by Todd Proebsting; we'd like to thank them for underwriting our vision.

References

Abelson, Harold, Sussman, Gerald Jay, & Sussman, Julie. (1985). *Structure and interpretation of computer programs*. Cambridge, Massachusetts: MIT Press.

Andreae, Chris, Noble, Jame, Markstrum, Shane, & Millstein, Todd. (2006). A framework for implementing pluggable type systems. *Pages 57–74 of: Proceedings of the 21st ACM SIGPLAN conference on object-oriented programing, systems, languages, and applications (OOPSLA'06)*. ACM Press.

Baader, Franz, & Nipkow, Tobias. (1998). *Term rewriting and all that*. New York, NY, USA: Cambridge University Press.

Bachrach, Jonathan, & Playford, Keith. (2001). The Java syntactic extender. *Pages 31–42 of: Proceedings of the 16th ACM SIGPLAN conference on object-oriented programming systems, languages and applications (OOPSLA '01)*. ACM Press.

Baker, Jason, & Hsieh, Wilson C. (2002). Maya: Multiple-dispatch syntax extension in Java. *Pages 270–281 of: Proceedings of the 2002 ACM SIGPLAN conference on programming language design and implementation (PLDI '02)*. ACM Press.

Barendregt, Henk. (1984). *The lambda calculus: Its syntax and semantics*. Revised edn. Studies in Logic and the Foundation of Mathematics, vol. 103. North-Holland.

Brabrand, Claus, Schwartzbach, Michael I., & Vanggaard, Mads. (2003). The Metafront system: Extensible parsing and transformation. *Electronic notes in theoretical computer science*, **82**(3), 592–611. Proceedings of LDTA 2003: Third Workshop on Language Descriptions, Tools and Applications.

Bravenboer, Martin, & Visser, Eelco. (2004). Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. *Pages 365–383 of:* Schmidt, Douglas C. (ed), *Proceedings of the 19th annual ACM SIGPLAN conference on object-oriented programing, systems, languages, and applications (OOPSLA'04)*. ACM Press.

Byrd, William E., & Friedman, Daniel P. 2006 (Sept.). From variadic functions to variadic relations: A miniKanren perspective. *Proceedings of the seventh workshop on Scheme and functional programming*. Technical Report TR-2006-06, Computer Science Department, University of Chicago.

Church, Alonzo. (1941). *The calculi of lambda-conversion*. Princeton University Press.

Clinger, William. (1991). Hygienic macros through explicit renaming. *LISP pointers*, **4**(4).

Clinger, William, & Rees, Jonathan. (1991). Macros that work. *Pages 155–162 of: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL '91)*. ACM Press.

Cousot, Patrick, & Cousot, Radhia. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Pages 238–252 of: Conference record of the fourth ACM symposium on principles of programming languages (POPL '77)*. ACM Press.

DeRemer, Frank, & Pennello, Thomas. (1982). Efficient computation of LALR(1) look-ahead sets. *ACM transactions on programming language systems*, **4**(4), 615–649.

Dybvig, R. Kent. (1992). *Writing hygienic macros in Scheme with syntax-case.* Tech. rept. TR 356. Indiana University Computer Science Department.

Dybvig, R. Kent, Hieb, Robert, & Bruggeman, Carl. (1992). Syntactic abstraction in Scheme. *LISP and symbolic computation*, **5**(4), 295–326.

Ekman, Torbjörn, & Hedin, Görel. (2004). Rewritable reference attributed grammars. *Pages 147–171 of: Proceedings of the 2004 European conference on object-oriented programming (ECOOP '04)*. Lecture Notes in Computer Science, vol. 3086. Springer.

Ekman, Torbjörn, & Hedin, Görel. (2007). The JastAdd extensible Java compiler. *Pages 1–18 of: Proceedings of the 22nd annual ACM SIGPLAN conference on object-oriented programming systems, languages and applications (OOPSLA '07)*. ACM Press.

Flatt, Matthew. (2002). Composable and compilable macros: you want it when? *Pages 72–83 of: Proceedings of the 7th ACM SIGPLAN international conference on functional programming (ICFP '02)*. ACM Press.

Flatt, Matthew, McMullan, Benjamin, Owens, Scott, & Shivers, Olin. 2004 (Oct.). Lexer and parser generators in Scheme. *Pages 41–52 of:* Shivers, Olin, & Waddell, Oscar (eds), *Proceedings of the 5th workshop on Scheme and functional programming (Scheme '04)*. Technical Report TR600, Computer Science Department, Indiana University.

Ford, Bryan. (2002). Packrat parsing: simple, powerful, lazy, linear time. *Pages 36–47 of: Proceedings of the seventh ACM SIGPLAN international conference on functional programming (ICFP 2002)*. ACM Press.

Friedman, Daniel P., Byrd, William E., & Kiselyov, Oleg. (2005). *The reasoned schemer*. Cambridge, Massachusetts: The MIT Press.

Grimm, Robert. (2006). Better extensibility through modular syntax. *SIGPLAN notices*, **41**(6), 38–51.

Herman, David, & Wand, Mitchell. (2008). A theory of hygienic macros. *Pages 48–62 of: Proceedings of the 17th European symposium on programming (ESOP 2008)*. Lecture Notes in Computer Science, vol. 4960. Springer.

ISO. (2004). *VRML ISO/IEC 14772 standard document*. Available at URL `http://www.web3d.org/x3d/specifications/vrml/`.

Johnson, Steven C. (1979). Yacc: Yet another compiler compiler. *Pages 353–387 of: UNIX programmer's manual*, vol. 2. New York, NY, USA: Holt, Rinehart, and Winston.

Krishnamurthi, Shriram, Erlich, Yan-David, & Felleisen, Matthias. (1999). Expressing structural properties as language constructs. *Pages 258–272 of: Proceedings of the 8th European symposium on programming languages and systems (ESOP '99)*. Lecture Notes in Computer Science, vol. 1576. Springer-Verlag.

Maddox, William. (1989). *Semantically-sensitive macroprocessing*. Tech. rept. CSD-89-545. University of California at Berkeley.

Mernik, Marjan, Žumer, Viljem, Lenič, Mitja, & Avdičaušević, Enis. (1999). Implementation of multiple attribute grammar inheritance in the tool LISA. *SIGPLAN notices*, **34**(6), 68–75.

Might, Matthew, & Shivers, Olin. (2006). Improving flow analyses via ΓCFA: Abstract garbage collection and counting. *Pages 13–25 of: Proceedings of the 11th ACM SIGPLAN international conference on functional programming (ICFP 2006)*. ACM Press.

Morrisett, Greg, Walker, David, Crary, Karl, & Glew, Neal. (1999). From system F to typed assembly language. *ACM transactions on programming language systems*, **21**(3), 527–568.

Nanavati, Ravi A. 2000 (September). *Extensible syntax in the presence of static analysis*. M.Phil. thesis, Massachusetts Institute of Technology.

Nystrom, Nathaniel, Qi, Xin, & Myers, Andrew C. (2006). J&: Nested intersection for scalable software composition. *Pages 21–36 of: Proceedings of the 21st ACM SIGPLAN conference on object-oriented programing, systems, languages, and applications (OOPSLA'06)*. ACM Press.

Olinsky, Reuben, Lindig, Christian, & Ramsey, Norman. (2006). Staged allocation: a compositional technique for specifying and implementing procedure calling conventions. *Pages 409–421 of: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL '06)*. ACM Press.

Pennello, Thomas J. (1986). Very fast LR parsing. *Pages 145–151 of: Proceedings of the 5th ACM SIGPLAN symposium on compiler construction (CC '86)*. ACM Press.

Peyton Jones, Simon, Eber, Jean-Marc, & Seward, Julian. (2000). Composing contracts: an adventure in financial engineering (functional pearl). *Pages 280–292 of: Proceedings of the 5th ACM SIGPLAN international conference on functional programming (ICFP '00)*. ACM Press.

Pottier, François, & Régis-Gianas, Yann. (2005). Towards efficient, typed LR parsers. *Pages 155–180 of: ACM workshop on ML*. Electronic Notes in Theoretical Computer Science, vol. 148, no. 2. Elsevier.

Pottier, Francois, & Remy, Didier. (2005). *The essence of ML type inference*. Cambridge, Massachusetts: The MIT Press. Pages 389–489.

Shalit, Andrew. (1996). *The Dylan reference manual*. Addison-Wesley.

Shivers, Olin. 1991 (May). *Control-flow analysis of higher-order languages, or taming lambda*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania. Technical Report CMU-CS-91-145.

Shivers, Olin. (2005). The anatomy of a loop: a story of scope and control. *Pages 2–14 of: Proceedings of the 10th ACM SIGPLAN international conference on functional programming (ICFP 2005)*. ACM Press.

Taha, Walid, & Sheard, Tim. (1997). Multi-stage programming with explicit annotations. *Pages 203–217 of: Proceedings of the 1997 ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation (PEPM '97)*. ACM Press.

Ungar, David, & Smith, Randall B. (1987). Self: The power of simplicity. *Pages 227–242 of: Proceedings of the 2nd ACM SIGPLAN conference on object-oriented programming systems, languages and applications (OOPSLA '87)*. ACM Press.

van Wyk, Eric, de Moor, Oege, Backhouse, Kevin, & Kwiatkowski, Paul. (2002). Forwarding in attribute grammars for modular language design. *Pages 128–142 of: Proceedings of the 11th international conference on compiler construction (CC '02)*. Lecture Notes in Computer Science, vol. 2304. Springer-Verlag.

van Wyk, Eric, Krishnan, Lijesh, Brodin, Derek, & Schwerdfeger, August. (2007). Attribute grammar-based language extensions for Java. *Pages 575–599 of: Proceedings of the 2007 European conference on object-oriented programming (ECOOP '07)*. Lecture Notes in Computer Science, vol. 4609. Springer-Verlag.

Visser, Eelco. (2004). *Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9*. Lecture Notes in Computer Science, vol. 3016. Spinger-Verlag. Pages 216–238.