A New Method for Dependent Parsing

Trevor Jim and Yitzhak Mandelbaum

AT&T Labs–Research

Abstract. *Dependent grammars* extend context-free grammars by allowing semantic values to be bound to variables and used to constrain parsing. Dependent grammars can cleanly specify common features that cannot be handled by context-free grammars, such as length fields in data formats and significant indentation in programming languages. Few parser generators support dependent parsing, however. To address this shortcoming, we have developed a new method for implementing dependent parsers by extending existing parsing algorithms. Our method proposes a point-free language of dependent grammars, which we believe closely corresponds to existing context-free parsing algorithms, and gives a novel transformation from conventional dependent grammars to point-free ones.

To validate our technique, we have specified the semantics of both source and target dependent grammar languages, and proven our transformation sound and complete with respect to those semantics. Furthermore, we have empirically validated the suitability of our point-free language by adapting four parsing engines to support it: an Earley parsing engine; a GLR parsing engine; memoizing, arrow-style parser combinators; and PEG parser combinators.

1 Introduction

Context-free grammars are widely used in data format and programming language specifications and are the foundation of many parsing tools. Unfortunately, they are not powerful enough to *fully* specify the syntax of most data formats and programming languages—these require context-sensitive features. For example, XML has balanced tags; many data formats have unbounded length fields; C and C++ have typedef names; Python, Haskell, and many markup languages have significant indentation; Javascript has optional line-ending semicolons; Standard ML has user-defined infix operators; and Ruby and command-line shells have "here documents."

Specifications that use grammars, therefore, augment them with prose describing the context-sensitive features of the syntax. This half-formal approach is not ideal. Often, it results in an ambiguous or incomplete specification, which leads to incompatible implementations. This problem is so severe that some communities have abandoned grammars altogether, e.g., the syntax of HTML5 is specified by a state machine given in pseudo-code [7].

Moreover, a specification given as a grammar plus prose cannot serve as the input to a parser generator or automated analysis. In the best case, the implementor will be able to figure out a "lexer hack" that hides context-sensitive

G. Barthe (Ed.): ESOP 2011, LNCS 6602, pp. 378–397, 2011.

features away from the grammar, which can then be processed separately by a parser generator. However, such tricks are hard to discover, and the result is not easy to understand, analyze, or replicate—witness the fact that there are many parser generators producing *parsers written in* Haskell or Python, but very few of them generate *parsers of* Haskell or Python.

In previous work, we used *dependent grammars* to cleanly specify context-sensitive syntax [10]. Dependent grammars extend context-free grammars by allowing semantic values to be bound to variables and used to guide subsequent parsing. For example, the value of a length field can be used to constrain the length of a following sequence, or the indentation of a line can be used to control the block structure of a Python program or Cisco IOS configuration section. We found dependent grammars to be an excellent formalism for specifying the kinds of context-sensitivity required in practical examples.

We also implemented dependent parsing, by extending Earley's algorithm for context-free parsing with semantic values, environments, and parsing constraints. This was more difficult. In particular, the machinery of environments had to be propagated throughout every part of the algorithm and correctness proof. This was delicate work, considering that there is a history of erroneous algorithms in the area (for example, Earley's algorithm for parse forest reconstruction did not account for all parse trees [17], and Tomita's original GLR algorithm fails to terminate on grammars with ϵ-rules and hidden left recursion [18]).

Dozens of other context-free parsing algorithms have been developed over many years, and we would like to adapt them for dependent parsing. Moreover, we would like to take advantage of existing *implementations* of these algorithms, because some of them have been finely tuned and represent many man-years of work. It is worth mentioning that, often, these sophisticated parsing engines do not even operate directly on grammars—instead, grammars are compiled into lower-level representations (for example, automatons), which are then "executed" by the parsing engine. Adding dependency to these engines by our previous method would involve complex changes to both the front-end and back-end, a prohibitive amount of work.

Therefore, we have developed a much simpler method of extending existing parsing algorithms to perform dependent parsing. Its key characteristics are:

- We introduce a new grammar *intermediate language* that supports dependent parsing without requiring environment machinery in the parsing engine.
- We compile a user-level dependent grammar into the intermediate language by a series of *source-to-source transformations* which move all environment manipulations into semantic actions of the intermediate language.

While there is a wealth of prior work on compiling away environment manipulations, from combinatory algebras to more recent work like Paterson's arrow notation [14], our work is distinguished by being compatible with a wide variety of existing parsing algorithms and engines (see Section 8 for further discussion). To demonstrate this compatibility, we have built four back ends for our dependent parser generator, each based on a different context-free parsing algorithm.

Contributions. We show how to use standard programming language and compiler techniques to implement parsers for dependent grammars:

- We define the semantics of Gul, a minimal user-level language of dependent grammars. Gul supports binding semantic values and using them in parse constraints, as well as standard semantic actions. Gul bindings are lexically scoped.
- We define the semantics of Gil, a point-free intermediate language of dependent grammars. Gil grammars parse inputs while passing semantic values from left to right (like an L-attributed grammar with guarded reductions), and, in our experience, it can be supported by most existing parsing engines with little difficulty.
- We define a novel source-to-source transformation for splitting a Gul grammar into (1) a Gil grammar and (2) a coroutine for managing binding and executing semantic actions.
- We have validated our choice of features in Gul by using it to implement grammars taken from a wide variety of domains.
- We have validated our technique by implementing Gil with a variety of different parsing backends, either through extension or directly on top of native features. These backends include a scannerless Earley parser, a GLR parser, arrow-style parser combinators, and PEG parser combinators.
- Finally, we have proven that our translation from Gul to Gil is semantics-preserving. The paper includes the statement of our central theorem, and a number of significant supporting lemmas, along with brief summaries of their proofs.

An extended version of the material in this paper can be found in a companion technical report [9].

2 Gul, a User-Level Language

Gul is the user-level dependent grammar language that we will use throughout the paper. Gul is a minimal language that omits many features of our parser generator, Yakker; however, Yakker itself implements most of these features by translation to Gul. Most of the features of Gul will be familiar to anyone who has used a lexer or parser generator. The more unusual aspects are these:

- We support all context free grammars, even ambiguous grammars. We do not require grammars to avoid left recursion, or be in a restricted class such as the LALR(1) or LL(k) grammars.
- We include parsing constraints, @when(e), which act to prune possible parses.
- We support *foreign parsers* with the form @box(e). For example, e could be a library function for parsing one of the hundreds of existing time and date formats.

Gul Syntax. We define Gul's syntax as follows:

$$G = (A_1(x_1) = R_1), \ldots$$
$$R = \epsilon \mid c \mid (R \mid R) \mid ({}^{*x=e}R) \mid (x{=}R\ R) \mid \{e\} \mid A(e) \mid \text{@when}(e) \mid \text{@box}(e)$$

A Gul *grammar* is a sequence of definitions for *nonterminals* in terms of *right sides*. We use G, A, and R to range over Gul grammars, nonterminals, and right sides, respectively. Gul right sides are based on regular expressions, including the empty string ϵ, *terminals* c, alternation, and Kleene closure (written as a prefix '*'). The Kleene closure performs a fold over a portion of the input. The expression e provides an initial value, and right-side R plays the role of a combining function: x is bound in R's scope to the previous accumulating value and R's result provides the new value. The expression e is taken from some general-purpose programming language, which we call the *target* language. The right side $(x{=}R_1\ R_2)$ is the concatenation of R_1 and R_2, where x is bound to the semantic value of R_1 in the scope R_2. *Semantic actions* are written $\{e\}$. Gul nonterminals are defined with formal parameters ranging over semantic values, and are applied to target-language expressions. Parsing constraints are written @when(e), and foreign parsers are written @box(e).

Our techniques are not specific to any particular target language, but for the sake of concreteness we will start by assuming some variant of the untyped, call-by-value lambda calculus.

Notational Conveniences. In our examples, we will write concrete terminals in quotes, for example, **'b'** for the ASCII character lowercase b. We assume that the target language has a distinguished unit value, written (), as well as booleans and some notion of sets (used in the semantics of @box(e)). If the parameter of a nonterminal A is not used in its right side, we omit it, and similarly we write $(R_1\ R_2)$ for a concatenation which does not require binding. A Kleene-closure which accumulates the unit value can be written $(*R)$. We may omit the parentheses in $(*R)$, $(*^{x=e}R)$, $(R_1\ R_2)$, $(R_1 \mid R_2)$, and $(x{=}R_1\ R_2)$ when this does not cause confusion. We write @pos for a @box that evaluates to the current input position, without consuming any input.

Gul Semantics. We now discuss the unusual aspects of Gul's semantics (we give the complete semantics in the technical report [9]). The semantics is defined by rules assuming a fixed grammar, G, and a fixed input, D, and they make use of the target language semantics via a partial evaluation function, $\text{eval}(E, e)$, where E is an environment mapping variables to values.

The rules define judgments of the form $\langle E, i\rangle \overset{R}{\Longrightarrow} \langle v, i'\rangle$, meaning that right side R evaluates to semantic value v in environment E, starting at input position i and finishing at position i'. Evaluation can be nondeterministic: we can have $\langle E, i\rangle \overset{R}{\Longrightarrow} \langle v_1, i_1\rangle$ and $\langle E, i\rangle \overset{R}{\Longrightarrow} \langle v_2, i_2\rangle$ where $v_1 \neq v_2$, or $i_1 \neq i_2$, or both.

These three rules show that we use a standard call-by-value semantics:

$$\frac{\text{eval}(E, e) = v}{\langle E, i\rangle \overset{\{e\}}{\Longrightarrow} \langle v, i\rangle} \qquad \frac{\langle E, i\rangle \overset{R_1}{\Longrightarrow} \langle v_1, i_1\rangle \quad \langle E[x = v_1], i_1\rangle \overset{R_2}{\Longrightarrow} \langle v_2, i_2\rangle}{\langle E, i\rangle \overset{(x=R_1\ R_2)}{\Longrightarrow} \langle v_2, i_2\rangle} \qquad \frac{\text{eval}(E, e) = v, \quad (A(x) = R) \in G \quad \langle [x = v], i\rangle \overset{R}{\Longrightarrow} \langle v_1, i_1\rangle}{\langle E, i\rangle \overset{A(e)}{\Longrightarrow} \langle v_1, i_1\rangle}$$

A semantic action $\{e\}$ evaluates to the value of e without consuming input; a concatenation $(x{=}R_1\ R_2)$ evaluates R_1, binds its value to x, then evaluates R_2; and $A(e)$ evaluates e, binds it to the formal parameter of A, then parses according to the right side of A.

The semantics of parsing constraints and foreign parsers are given as follows:

$$\frac{\text{eval}(E, e) = \text{true}}{\langle E, i\rangle \xrightarrow{@\text{when}(e)} \langle (), i\rangle} \qquad \frac{\langle v, j\rangle \in \text{eval}(E, e(D)(i))}{\langle E, i\rangle \xrightarrow{@\text{box}(e)} \langle v, j\rangle}$$

The rule for @when(e) states that no progress can be made unless the constraint e is satisfied (there is no rule for the false case). To evaluate @box(e), we evaluate e applied to the complete input D and the current input position i. The foreign parser can be nondeterministic—it may return more than one result $\langle v, j\rangle$—but we require $j \geq i$. Note that the expression e can include values bound within the grammar (e.g., by (x=R_1 R_2)). Furthermore, a foreign parser is free to examine the complete input, and not just the portion between i and j. Note that foreign parsers subsume semantic actions, parsing constraints, terminals, and the empty string. Similarly, Kleene closure could be encoded using an additional nonterminal. Howeve, we have chosen to include those forms natively for reasons discussed at the end of Section 3.

We say that *a grammar G accepts input D with value v* if $\langle \cdot, 0\rangle \xRightarrow{A_1} \langle v, |D|\rangle$, where A_1 is the implicit start symbol of G. While we restrict our definition to parses which begin with an empty environment, in practice, we expect that grammars will be written with respect to some distinguished environment E_{init}, which may contain bindings for library functions, foreign parsers, etc. We can simulate such an environment simply by substituting the contents of E_{init} into our grammar before parsing.

We give an example Gul grammar in Section 4.2.

3 The Intermediate Language Gil

Gil is a lower-level language that corresponds closely to context-free grammars extended to support semantic values.

Gil Syntax. We define Gil's syntax as follows:

$g = (A_1 = r_1), \ldots$
$r = \epsilon \mid c \mid (r \mid r) \mid (^*r) \mid (r\ r) \mid \{f\} \mid A(f_{\text{arg}}, f_{\text{ret}}) \mid @\text{when}(f_{\text{pred}}, f_{\text{next}}) \mid @\text{box}(f_{\text{box}}, f_{\text{ret}})$

Gil, like Gul, is based on regular expressions over terminals and nonterminals. To distinguish between Gul and Gil we use g, r, and f to range over Gil grammars, right sides, and target language expressions, instead of Gul's G, R, and e. Gil lacks the binding forms of Gul: nonterminals are defined without a formal parameter, and there is no binding concatenation. Note that in Gil, nonterminals, constraints and foreign parsers take two arguments, whose purpose we will explain in a moment.

Gil Semantics. We give the full semantics of Gil in the technical report [9], and note a few key points here in the main text. Gil right sides are semantic-value *transformers*—they relate input values to output values—and thus, each right side takes an *implicit* value parameter. Semantic actions are the base value

transformers, and our rule for concatenation shows that Gil threads values from left to right across parses:

$$\frac{f(v) = v_1}{\langle v, i\rangle \xrightarrow{\{f\}} \langle v_1, i\rangle} \qquad \frac{\langle v, i\rangle \xrightarrow{r_1} \langle v_1, i_1\rangle,\ \langle v_1, i_1\rangle \xrightarrow{r_2} \langle v_2, i_2\rangle}{\langle v, i\rangle \xrightarrow{(r_1 r_2)} \langle v_2, i_2\rangle}$$

Here are the rules for parsing constraints, nonterminals, and foreign parsers:

$$\frac{f_{\text{pred}}(v) = \text{true} \quad (v' = f_{\text{next}}(v))}{\langle v, i\rangle \xrightarrow{@\text{when}(f_{\text{pred}}, f_{\text{next}})} \langle v', i\rangle} \qquad \frac{f_{\text{arg}}(v) = v_1, (A = r) \in g \quad \langle v_1, i\rangle \xrightarrow{r} \langle v_2, i_2\rangle \quad f_{\text{ret}}(v)(v_2) = v_3}{\langle v, i\rangle \xrightarrow{A(f_{\text{arg}}, f_{\text{ret}})} \langle v_3, i_2\rangle} \qquad \frac{\langle v_1, i_1\rangle \in f_{\text{box}}(v)(D)(i) \quad f_{\text{ret}}(v)(v_1) = v_2}{\langle v, i\rangle \xrightarrow{@\text{box}(f_{\text{box}}, f_{\text{ret}})} \langle v_2, i_1\rangle}$$

The first argument of a parsing constraint is used to compute a boolean determining whether parsing will continue, while the second is for calculating the transformed value. Nonterminals and boxes use their first argument to calculate the argument of the nonterminal or box, and the second to merge the result value with the original value. For example, if we define

$$\text{zero} = \textbf{`0'}\ \{\lambda v.2 \times v\}, \qquad \text{one} = \textbf{`1'}\ \{\lambda v.(2 \times v) + 1\},$$

then we can calculate the binary value of a sequence of 1s and 0s with

$$\text{bits} = \{\lambda v.0\}\ {}^*(\text{zero}(\lambda v.v, \lambda v.\lambda v_2.v_2) \mid \text{one}(\lambda v.v, \lambda v.\lambda v_2.v_2)).$$

Here we use the action $\{\lambda v.0\}$ to initialize the value to 0. The function $\lambda v.v$ used in the argument positions for zero and one simply propagates the current value to those nonterminals. The function $\lambda v.\lambda v_2.v_2$ is used on the return, and it sets the new current value to the value v_2 returned from the nonterminal.

Finally, we can calculate a sum with this right side:

$$\text{bits}(\lambda v.(), \lambda v.\lambda v_1.v_1)\ \textbf{`+'}\ \text{bits}(\lambda v_1.(), \lambda v_1.\lambda v_2.v_1 + v_2).$$

Here the value returned from the first parse of bits is bound to v_1, and the value from the second parse of bits is bound to v_2. The function $\lambda v_1.\lambda v_2.v_1 + v_2$ performs the addition. Notice that there are two occurrences of (λv_1), one for each parse of bits; the semantics of nonterminals ensures that all occurrences of v_1 end up bound to the same value.

From these examples, we can see that the second argument of a nonterminal acts something like a continuation. Unlike in continuation-passing style, however, the continuation is *not* passed to the right side of the nonterminal; it is invoked by the caller, and not the callee. This is necessary to achieve maximal sharing to efficiently parse ambiguous grammars, which may require multiple parses of a single nonterminal at the same input position and with the same input parameter, but with different continuations.

In Gil, foreign parsers could subsume the empty string, terminals, and parsing constraints, and Kleene closure could be encoded using an additional nonterminal. We have retained these features because they are either supported natively by existing context-free parsing engines or could be more efficiently implemented on top of existing features than the more general @box.

4 The Coroutine Transformation

In this section, we show how to compile a Gul rule with bindings, parsing constraints, and other semantic actions into a Gil rule in which all of the semantic elements of the Gul rule have been gathered together into one Gil action (value transformer). This single Gil action is used as a sort of coroutine by the parsing engine as it processes the input according to the rest of the Gil rule.

4.1 Assumptions on the Target Language

The coroutine is a target language program that we build from the actions in the original Gul rule. We treat these actions opaquely, in a cut-and-paste fashion; consequently, we limit our assumptions about the target language and make our techniques more widely applicable. Nevertheless, to build the coroutine, we will have to assume that some language features are available. Essentially, we require the target language to support some features of an untyped call-by-value lambda calculus, as indicated by the following grammar of expressions:

$$\begin{aligned} e &= x \mid (e\ e) \mid (\lambda x.e) \mid (\text{fun}\, f\, x.e) \mid () \mid \ell \mid \underline{\lambda}\, C \mid \ldots \\ C &= \ell.e \mid (C \mid C) \end{aligned}$$

We are assuming that we can use target language variables and function application, that we have first-class and recursive functions, and we have a distinguished unit value (). We assume that we can use a countable set of *labels*, ranged over by ℓ; for concrete labels, we use underlined integers, like $\underline{3}$. We also make use of a match-function construct: $\underline{\lambda}(\ell_1.e_1 \mid \cdots \mid \ell_n.e_n)$. Here, we expect that a match-function is applied to a label ℓ_i, and the result of the application is the corresponding case e_i.[1] Finally, we permit let expressions which can be desugared in the standard way.

Note that we have not said that the target language *is* an untyped lambda calculus—it is sufficient for these features to be embeddable in the target language. In Section 5, we discuss the exact properties that we require of the target language. When the target language is statically typed, there are additional considerations, which we discuss in Section 6.

4.2 Coroutines by Example

To illustrate Gul-to-Gil compilation, we use an example adapted from the grammar for the IMAP mail protocol:

$$\text{literal} = \text{`\{'}\ x{=}\text{number}\ \ \text{`\}'}\ p_1{=}@\text{pos}\ \ {*}\text{CHAR8}\ p_2{=}@\text{pos}\ \ @\text{when}(p_2 - p_1 = x)$$

An IMAP literal is a number surrounded by braces, followed by a sequence of characters (CHAR8s). We assume that number is a nonterminal that matches a

[1] As a corollary of our correctness proof, we know that we cannot have match failures in our construction.

sequence of ASCII digits and returns the equivalent semantic integer as its result. Recall that @pos is an abbreviation for a foreign parser that matches the empty string and returns the input position, so that p_1=@pos binds p_1 to the position in the input just after the right brace, and p_2=@pos binds p_2 to a position after some number of CHAR8s. The parse constraint @when$(p_2 - p_1 = x)$ matches the empty string if the predicate $(p_2 - p_1 = x)$ returns true, and otherwise fails to match anything.[2]

We transform this Gul rule into the following Gil rule:

$$\text{literal} = \{f_{\text{init}}\}\ \textbf{`\{'}\ \text{number}(f_1, f_2)\ \textbf{`\}'}\ \{f_3\}\ \text{*CHAR8}\ \{f_4\}\ @\text{when}(f_5, f_6)$$

(We will define f_{init} and $f_{1\text{–}6}$ shortly.)

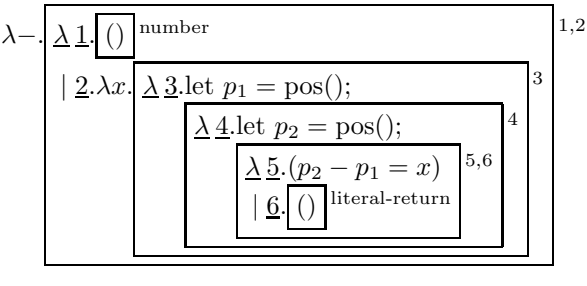
Recall that every Gil rule is a value transformer, and the initial value of a rule is its (implicit) parameter. Here, literal has no parameter, so we will use the unit value () for its initial value. According to the Gil semantics and the right side of the rule, we can begin parsing a literal by passing the initial value to f_{init}, which is itself a value transformer that we define as follows:

$$\begin{aligned} f_{\text{init}} = \lambda{-}.&\underline{\lambda}\,\underline{1}.() \\ &\mid \underline{2}.\lambda x.\underline{\lambda}\,\underline{3}.\text{let } p_1 = \text{pos}(); \\ &\qquad \underline{\lambda}\,\underline{4}.\text{let } p_2 = \text{pos}(); \\ &\qquad\quad \underline{\lambda}\,\underline{5}.(p_2 - p_1 = x) \\ &\qquad\quad \mid \underline{6}.() \end{aligned}$$

As we will see, f_{init} is the coroutine that carries out all of the semantic actions of the original Gul rule.

The Gil parsing engine starts parsing the rule by applying f_{init} to the current value, and expects to get a new value in return. f_{init} takes the current value and ignores it $(\lambda{-})$, since it is unit. f_{init} returns a new value (beginning with $\underline{\lambda}\,\underline{1}$) that incorporates all of the semantic actions of the original Gul rule. The parser will pass this value through a series of the other transformers $f_{1\text{–}5}$ of the rule to execute the semantic actions as required.

It is helpful to annotate f_{init} to highlight the values that will be passed to each transformer by the parser:



[2] IMAP literals can be implemented more efficiently in Gul, but this simple version serves better for our exposition. Also, this example can be generalized to support repetition of arbitrary-length nonterminals by including a counter variable on the Kleene-closure and comparing it to x in the constraint.

Here, we have boxed each expression that will compute a return value for a transformer. The label of a box indicates which transformer or transformers will receive the value. For example, f_{init} returns a value that will be passed to f_1 and f_2, so we box the return value and label it with $1, 2$.

Parsing continues by reading a left brace character, reaching number(f_1, f_2) in the right side. We apply f_1 to the current value (the 1,2-box) to calculate the parameter to number. We define f_1 simply as $\lambda v.v(\underline{1})$, so we end up applying the value to $\underline{1}$, hence the $(\underline{\lambda}\,1)$ case of the 1,2-box is evaluated. This returns (), since number takes no parameters; we have boxed this and labeled it to indicate that it will be the initial value passed to number.

The parsing engine uses f_2 to handle the binding of x to the result of number. We define $f_2 = \lambda v.v(\underline{2})$, so we end up applying the 1,2-box to $\underline{2}$, and hence the $(|\ \underline{2})$ case is evaluated. The parsing engine applies this to the result of number. The number is bound to x and the 3-box becomes the new current value.

Next, the parsing engine applies f_3 to the current value. We define $f_3 = \lambda v.v(\underline{3})$, so the current input position is bound to p_1, and the 4-box becomes the new value.

$f_{4\text{–}6}$ are defined in the same way as $f_{1\text{–}3}$, so the parsing engine ends up using the 4-box to bind p_2, and the 5,6-box to calculate the predicate $(p_2 - p_1 = x)$. If this evaluates to true, the engine uses the 5,6 box to calculate the final value, (), of the successful parse; otherwise, this run of the parse fails.

4.3 Coroutines Formalized

We now formalize the translation of Gul into Gil. We begin with a property that conservatively approximates when nonterminals and right sides make use of Gul's context-sensitive features. We term this property *relevance*.

Definition 1 (Relevance). *The relevance of the nonterminals and right sides of a grammar are defined as the least relations satisfying the following properties:*

- *A right side is* relevant *if it includes a target-language expression or a* relevant *nonterminal.*
- *A nonterminal is* relevant *if its right-side is* relevant.

Notice that bindings do not impact relevance, because what matters for parsing is whether the binding is used.

Irrelevant and relevant right-sides are handled differently by our translation. Therefore, to reduce the number of cases that need be considered during the translation, we specify a normal form for grammars that places syntax-directed constraints on the relevance of subterms. Normalized grammars use an extended syntax that includes the forms $(R_1\ R_2)$, $(^*R)$ and A, which are only abbreviations in Gul.

Definition 2 (Normalization). *A right side R is* normalized *if every subterm R' of R satisfies the following properties:*

N1 *If R' is $(x{=}R_1\ R_2)$ then both R_1 and R_2 are relevant.*
N2 *If R' is $(R_1\ R_2)$ then at least R_1 is not relevant.*

N3 *If R' is $(R_1 \mid R_2)$ then R_1 and R_2 share the same relevance.*
N4 *If R' is $(*R_1)$ then R_1 is not relevant.*
N5 *If R' is $({}^{*x=e}R_1)$ then R_1 is relevant.*
N6 *If R' is A then the right-side defining A in G is not relevant.*
N7 *If R' is $A(e)$ then the right-side defining A in G is relevant.*

A grammar G is normalized *if every rule satisfies the following properties:*

N8 *If $(A(x) = R) \in G$ then R is relevant.*
N9 *If $(A = R) \in G$ then R is not relevant.*

Any right side R can be transformed into a normalized right side accepting the same language, and similarly for any grammar. We illustrate such transformations in [9].

Once a grammar has been normalized, we can translate its rules into Gil as follows. If R is not relevant, then R only uses syntax common to both Gul and Gil—that is, R is a Gil right side. In that case, the Gul rule $A = R$ is translated to the Gil rule $A = R$. If R is relevant, then a Gul rule $A(x) = R$ is turned into a Gil rule by a sequence of transformations, as indicated in the following definition. Below, we describe each of the transformations in turn.

Definition 3 (Gul-to-Gil Transformation). *We say that a normalized Gul grammar G transforms to a Gil grammar g, written $G \Rightarrow g$, iff*

- *If $(A = R) \in G$ then $(A = R) \in g$.*
- *If $(A(x) = R) \in G$ then $(A = \{\lambda x.\underline{\lambda}(\mathcal{C}[\![\mathcal{E}[\![R_\ell]\!]]\!][\cdot])\}\ \mathcal{D}[\![R_\ell]\!]) \in g$, where $R_\ell = \mathcal{L}[\![R]\!]$.*

Labeling $\mathcal{L}[\![\cdot]\!]$. Our first step is to add *labels* to Gul right sides. These labels serve to synchronize the construction of coroutines with the insertion of dispatch functions. The insertion of labels considerably simplifies the specification of those two phases, which otherwise could not be specified independently.

We only need to add labels to relevant subterms of a right side. The labeling transformation is given in Figure 1. We use underlined integers for labels, and use ℓ to range over integers used as labels. Each label identifies a control-flow point in the right side: in ${}^{\ell}R$, ℓ is the control-flow point just before evaluating R, and in R^{ℓ}, ℓ is the control-flow point just after evaluating R. In the case for sequences, we do not label subterm R_1, because we are guaranteed, by normalization, that it is irrelevant.

Erasing $\mathcal{E}[\![\cdot]\!]$. The coroutine for a Gul right side is constructed exclusively from relevant subterms of the right side. We can simplify the definition of coroutine production if we first erase all subterms that are not relevant. A suitable transformation is given in Figure 2. It has the important property that the resulting right side exactly preserves the control-flow of the labels of the original right side. The interesting case is for sequences, which, by normalization, are the only place irrelevant terms may appear within relevant terms. Notice that the result of erasing looks like a labeled, nondeterministic program.

$\boxed{\mathcal{L}[\![R]\!] = R'}$ (R is normalized)

If R is not relevant, then $\mathcal{L}[\![R]\!] = R$.

Otherwise, $\mathcal{L}[\![R]\!]$ is defined by the following cases. In each case, ℓ and ℓ' denote fresh labels.

$$\mathcal{L}[\![\{e\}]\!] = {}^{\ell}\{e\}$$

$$\mathcal{L}[\![A(e)]\!] = {}^{\ell}A(e)^{\ell'}$$

$$\mathcal{L}[\![@\mathrm{box}(e)]\!] = {}^{\ell}@\mathrm{box}(e)^{\ell'}$$

$$\mathcal{L}[\![@\mathrm{when}(e)]\!] = {}^{\ell}@\mathrm{when}(e)^{\ell'}$$

$$\mathcal{L}[\![(R_1 \mid R_2)]\!] = (\mathcal{L}[\![R_1]\!] \mid \mathcal{L}[\![R_2]\!])$$

$$\mathcal{L}[\![(R_1\ R_2)]\!] = (R_1\ \mathcal{L}[\![R_2]\!])$$

$$\mathcal{L}[\![(x{=}R_1\ R_2)]\!] = {}^{\ell}(x{=}\mathcal{L}[\![R_1]\!]\ \mathcal{L}[\![R_2]\!])$$

$$\mathcal{L}[\![(*^{x=e}R)]\!] = {}^{\ell}(*^{x=e}\mathcal{L}[\![R]\!])^{\ell'}$$

Fig. 1. Labeling right sides

$\boxed{\mathcal{E}[\![R]\!] = R'}$ (R is relevant and normalized)

$$\mathcal{E}[\![{}^{\ell}\{e\}]\!] = {}^{\ell}\{e\}$$

$$\mathcal{E}[\![{}^{\ell}A(e)^{\ell'}]\!] = {}^{\ell}A(e)^{\ell'}$$

$$\mathcal{E}[\![{}^{\ell}@\mathrm{box}(e)^{\ell'}]\!] = {}^{\ell}@\mathrm{box}(e)^{\ell'}$$

$$\mathcal{E}[\![(R_1 \mid R_2)]\!] = (\mathcal{E}[\![R_1]\!] \mid \mathcal{E}[\![R_2]\!])$$

$$\mathcal{E}[\![(R_1\ R_2)]\!] = \mathcal{E}[\![R_2]\!]$$

$$\mathcal{E}[\![{}^{\ell}(x{=}R_1\ R_2)]\!] = {}^{\ell}(x{=}\mathcal{E}[\![R_1]\!]\ \mathcal{E}[\![R_2]\!])$$

$$\mathcal{E}[\![{}^{\ell}(*^{x=e}R)^{\ell'}]\!] = {}^{\ell}(*^{x=e}\mathcal{E}[\![R]\!])^{\ell'}$$

Fig. 2. Erasing irrelevant subterms

Coroutine Production $\mathcal{C}[\![\cdot]\!]$. Given a labeled Gul right side R, we construct the match-cases of its coroutine using the function $\mathcal{C}[\![\cdot]\!]$, defined in Figure 3. The function makes use of *contexts* defined by the grammar

$$K = [\cdot] \mid x\ ([\cdot]),$$

and we write $K[e]$ to "fill the hole" of K, resulting in a target expression. Our contexts are much simpler than typical contexts used in programming languages: they have only two forms, and they cannot bind variables. $\mathcal{C}[\![\cdot]\!]$ is written in the style of a continuation-passing transform, and we use contexts K as the continuation argument of the transform. By using contexts, rather than target language expressions, as arguments to the transform, we obtain a one-pass algorithm whose result does not have administrative redexes [3].

The first case of Figure 3 handles semantic actions. Unlike in Gul, the results of semantic actions are not passed to the parsing engine, but are instead passed directly to the context K. This directness has some helpful implications for typed target languages, which we discuss in Section 6. The case for constraints splits its arguments into two match-cases, one for each label. The first evaluates the predicate and the second (invoked when the predicate is true) evaluates the input continuation. In Gul, constraints always have unit as their semantic value, so we pass the continuation the unit value. The case for nonterminals is quite similar to that of constraints, but reifies the continuation as a function expecting the return value of the nonterminal. The case for box does the same.

$\boxed{\mathcal{C}[\![R]\!]K = C}$

$\mathcal{C}[\![^{\ell}\{e\}]\!]K = \ell.K[e].$

$\mathcal{C}[\![^{\ell}@\mathrm{when}(e)^{\ell'}]\!]K = \big(\ell.e \mid \ell'.K[()]\big).$

$\mathcal{C}[\![^{\ell}A(e)^{\ell'}]\!]K = \big(\ell.e \mid \ell'.\lambda x.K[x]\big)$
where x is a fresh variable.

$\mathcal{C}[\![^{\ell}@\mathrm{box}(e)^{\ell'}]\!]K = \big(\ell.e \mid \ell'.\lambda x.K[x]\big)$
where x is a fresh variable.

$\mathcal{C}[\![(R_1 \mid R_2)]\!]K = \big(\mathcal{C}[\![R_1]\!]K \mid \mathcal{C}[\![R_2]\!]K\big).$

$\mathcal{C}[\![^{\ell}(x{=}R_1\ R_2)]\!]K =$
$\quad \ell.\mathrm{let}\ g\ x = \underline{\lambda}\big(\mathcal{C}[\![R_2]\!]K\big);$
$\quad\quad \underline{\lambda}\big(\mathcal{C}[\![R_1]\!](g[\cdot])\big)$

$\mathcal{C}[\![^{\ell}(*^{x=e}R)^{\ell'}]\!]K =$
$\quad \ell.\mathrm{let\ rec}\ g\ x = \underline{\lambda}\big(\ell'.K[x] \mid \mathcal{C}[\![R]\!](g[\cdot])\big);$
$\quad\quad g(e)$

Fig. 3. Coroutine production. In the last two cases, g denotes a fresh variable.

$\boxed{\mathcal{D}[\![R]\!] = r}$ (R is normalized)

If R is not relevant, then $\mathcal{D}[\![R]\!] = R$.

Otherwise, $\mathcal{D}[\![R]\!]$ is defined as follows, where $\mathrm{d}(\ell) = \lambda v.v(\ell)$:

$\mathcal{D}[\![^{\ell}\{e\}]\!] = \{\mathrm{d}(\ell)\}$

$\mathcal{D}[\![^{\ell}@\mathrm{when}(e)^{\ell'}]\!] = @\mathrm{when}(\mathrm{d}(\ell), \mathrm{d}(\ell'))$

$\mathcal{D}[\![^{\ell}@\mathrm{box}(e)^{\ell'}]\!] = @\mathrm{box}(\mathrm{d}(\ell), \mathrm{d}(\ell'))$

$\mathcal{D}[\![^{\ell}A(e)^{\ell'}]\!] = A(\mathrm{d}(\ell), \mathrm{d}(\ell'))$

$\mathcal{D}[\![(R_1 \mid R_2)]\!] = (\mathcal{D}[\![R_1]\!] \mid \mathcal{D}[\![R_2]\!])$

$\mathcal{D}[\![(R_1\ R_2)]\!] = (R_1\ \mathcal{D}[\![R_2]\!])$

$\mathcal{D}[\![^{\ell}(x{=}R_1\ R_2)]\!] = \{\mathrm{d}(\ell)\}\ \mathcal{D}[\![R_1]\!]\ \mathcal{D}[\![R_2]\!]$

$\mathcal{D}[\![^{\ell}(*^{x=e}R)^{\ell'}]\!] = \{\mathrm{d}(\ell)\}\ *(\mathcal{D}[\![R]\!])\ \{\mathrm{d}(\ell')\}$

Fig. 4. The dispatching transformation

In the case of choice, we simply combine the match-cases of each branch into a single set of match cases. In the case for binding, we reify the continuation for R_1 as a (bound) function before transforming R_1. This detail prevents the inadvertent capture of free variables in R_2 by bindings in R_1. Since we are treating the semantic actions in Gul grammars opaquely (Section 4.1), we do not have the luxury of alpha-varying right-sides. We do, however, assume the ability to generate variables which are fresh with respect to target-language expressions.

Finally, the case for fold performs a reification just as in the binding case. The coroutine uses a recursive function, and combines the case for exiting the fold with the cases of the fold body. We provide an initial value to x by applying the function to expression e, and bind x to the body's most recent semantic value in each iteration of the function.

Dispatching $\mathcal{D}[\![\cdot]\!]$. We transform the labeled Gul right side to a Gil right side by replacing each label occurrence ℓ with a *dispatch* function ($\lambda v.v(\ell)$), which "informs" the coroutine of the current position in the control-flow graph. The transformation is given in Figure 4. As with labeling, we ignore R_1 in the case for sequences.

5 Correctness

The goal of our coroutine transformation is to simplify both the task of implementing a dependent parsing system and the task of proving said system correct. This section discusses correctness: specifically, that the result of our transformation parses the same language and results in the same value as the original Gul grammar, under appropriate conditions. We provide a summary of our correctness results here, and leave an extended discussion to our companion technical report [9].

An essential goal of our construction has been to avoid dealing with the details of the target language, as we discussed earlier. Our metatheory retains this focus, by stating and proving our theorem with respect to an *abstract* target language. Instead of a concrete language, we have derived a set of properties that we require of the evaluation function of the language in order to prove our soundness and completeness result. These properties are given in [9], and are unexceptional.

In the theorem and lemmas that follow, we use some well-formedness conditions for grammars, expressions, contexts and right-sides. These conditions essentially ensure that variables and nonterminal names are properly bound. Only well-formedness of contexts requires, in addition, *totality* from the context: if its hole is filled with a closed value, it must evaluate to a value. The formal definitions of well-formedness can be found in [9].

It is worth noting that we require call-by-value evaluation for the function constructs used by coroutines. This constraint is needed for soundness because we have defined Gul with a call-by-value semantics; a call-by-name semantics for coroutines would result in some Gil parses succeeding where Gul parses fail. We believe it would be straightforward to adapt Gul, Gil, and our transformation to a call-by-name semantics.

Theorem 1, below, states the main result of this section.

Theorem 1. *If G is normalized, $WF(G)$, and $G \Rightarrow g$, then*

1. *If $(A = R) \in G$ and $(A = r) \in g$, then for all inputs,*
   (a) if $\langle \cdot, 0 \rangle \xRightarrow{R} \langle v, i \rangle$ then $\langle (), 0 \rangle \xrightarrow{r} \langle (), i \rangle$, and
   (b) if $\langle (), 0 \rangle \xrightarrow{r} \langle (), i \rangle$ then $\langle \cdot, 0 \rangle \xRightarrow{R} \langle (), i \rangle$.
2. *If $(A(x) = R) \in G$ and $(A = r) \in g$, and v closed, then for all inputs,*
   (a) if $\langle [x{=}v], 0 \rangle \xRightarrow{R} \langle v', i \rangle$ then $\langle v, 0 \rangle \xrightarrow{r} \langle v', i \rangle$, and
   (b) if $\langle v, 0 \rangle \xrightarrow{r} \langle v', i \rangle$ then $\langle [x{=}v], 0 \rangle \xRightarrow{R} \langle v', i \rangle$.

In order to prove Theorem 1, we will state and prove stronger results for both relevant and irrelevant terms. We need the stronger statements because the theorem describes properties of whole rules, while we need to know properties of individual right-sides. Yet, this change is not as easy as it might sound. The coroutine transformation is not local to right sides, but global to an entire rule. Therefore, we must relate the coroutine that would be generated for a particular right-side in *isolation* to the coroutine of the rule of which the right side is part.

A first attempt might be to relate the coroutines with equivalence. However, equivalence alone is too strict for relating our local and global coroutines because

the case for alternatives and fold both locally require a *smaller* coroutine—one with fewer match cases—than that provided by the surrounding context. Therefore, we extend equivalence to an ordering relation on expressions, similar to subtyping. We call this relation *sufficiency*, written $E \vdash e \Rightarrow e'$, and read "$e$ is sufficient for e'."

Informally, an expression e is sufficient for an expression e' if e can be used in place of e'. In the study of subtyping, this is often called the *principal of safe substitution* [16]. We capture this notion formally in the following lemma, which we use throughout our proof of Lemma 3, below.

Lemma 1. *Let* $e = \underline{\lambda}(\ell_1.e_1 \mid \ldots \mid \ell_n.e_n)$. *If* $WF(E, e)$ *and* $E \vdash e' \Rightarrow e$ *then* $E \vdash e'\ \ell_i \equiv e\ \ell_i$, *for* $i = 1$ *to* n.

We now state the two essential lemmas used in proving the soundness and completeness of our transformation of Gul grammars. Note that both of these lemmas assume an implicit Gul grammar G, which is normalized and well-formed, and a corresponding, implicit, Gil grammar g, for which $G \Rightarrow g$ holds. The first lemma addresses irrelevant right sides, on which the translation has little effect.

Lemma 2 (Translation correctness for irrelevant terms). *If* R *is irrelevant and normalized, then*

a) If $\langle E, i\rangle \overset{R}{\Longrightarrow} \langle v', i'\rangle$ *then* $\forall v,\ \langle v, i\rangle \overset{R}{\longrightarrow} \langle v, i'\rangle$.
b) If $\langle v, i\rangle \overset{R}{\longrightarrow} \langle v', i'\rangle$ *then* $v = v'$ *and* $\forall E,\ \langle E, i\rangle \overset{R}{\Longrightarrow} \langle (), i'\rangle$.

The second lemma addresses relevant right sides. Notice how we relate the current Gil value v_1 with the right-side's coroutine via the sufficiency relation. Notice further, though, that our result involves *equivalence*. This apparent strengthening is due to Lemma 1 above.

Most of the proof of this lemma involves reasoning about the evaluation behavior of the generated coroutines.

Lemma 3 (Translation correctness for relevant terms). *If* R *is relevant and normalized,* $WF(E,\ R)$, $WF(E,\ K)$, v_1 *is closed,* $R_\ell = \mathcal{L}[\![R]\!]$, *and* $E \vdash v_1 \Rightarrow \underline{\lambda}(\mathcal{C}[\![\mathcal{E}[\![R_\ell]\!]]\!]K)$ *then*

a) If $\langle E, i\rangle \overset{R}{\Longrightarrow} \langle v, i'\rangle$ *then* $\exists v_2.\ \langle v_1, i\rangle \xrightarrow{\mathcal{D}[\![R_\ell]\!]} \langle v_2, i'\rangle$ *and* $E \vdash K[v] \equiv v_2$.
b) If $\langle v_1, i\rangle \xrightarrow{\mathcal{D}[\![R_\ell]\!]} \langle v_2, i'\rangle$ *then* $\exists v.\ \langle E, i\rangle \overset{R}{\Longrightarrow} \langle v, i'\rangle$ *and* $E \vdash K[v] \equiv v_2$.

6 Typed Target Languages

Up until now we have assumed that our target language is untyped. However, we have implemented our parser generator, Yakker, in a statically typed language (OCaml), and this requires a few modifications to our coroutine transformation.

The principal difficulty is that many parsing engines need to manipulate collections of semantic values, for example, on a semantic-value stack. ML's homogeneous data structures therefore require us to give our semantic values

(coroutines) a uniform type. Since our coroutines return a number of types (booleans for parsing constraints, foreign parsers, etc.), we must wrap them in a union datatype.

This sort of type casting is standard in ML parsers. The coroutine transformation significantly simplifies matters, however, because the types of Gul-bound variables do not appear in the coroutine union type: all bindings are implemented by closures, which hide the types of free variables. This is a key reason that we prefer coroutines over our original implementation of dependent parsing that used explicit environments.

We use a separate union type for call arguments and return values, which we call the *value* type. In Yakker, we currently require user annotations to specify argument and return types, and we construct the value type and insert the necessary injections and projections automatically. (The type annotations are not strictly necessary, e.g., the dypgen parser generator is able to eliminate them by type inference.) The coroutine type is then given as follows:

```
type value
type coroutine =
  | Bool of boolean
  | Value of value
  | Return of coroutine → coroutine
  | Box of int → input → (coroutine ∗ int) list
  | Continue of int → coroutine
```

Next, we must add the necessary injections and projections to and from the coroutine type. We add injections in the translation from Gul to the coroutine, and we add projections in the translation from Gul to Gil. The addition of injections is largely straightforward, and we only note that every nonterminal must end with a Value injection, which can be accomplished by setting the context used to construct the initial coroutine to Value[·].

Projections are similarly straightforward. First, change dispatch functions to project from the Continue branch and then compose every dispatch with a projection appropriate to the location of the dispatch. So, for example, a dispatch located in a @box would be followed by a projection from the Box branch. Second, the initial coroutine must begin by projecting its argument from the Value branch, as must all f_{ret} functions. For example, here is the typed rule for nonterminals:

$$\mathcal{C}[\![^{\ell}A(e)^{\ell'}]\!]K = \big(\ell.\text{Value}(e) \mid \ell'.\lambda\text{Value}(x).K[x]\big) \text{ where } x \text{ is fresh.}$$

Notice that the first match-case performs an injection, whereas the second case uses pattern matching to perform a projection.

7 Evaluation

We have investigated several practical aspects of our method.

Implementing Gil. To evaluate the difficulty of extending a context-free parsing algorithm to support the additional features of Gil, we implemented Gil on four different back ends:

Scannerless Earley. Our main implementation is a transducer-based, scannerless Earley parsing engine. Earley parsing is a general context-free parsing method that relies on nondeterministic, breadth-first exploration of possible parses. The standard Earley algorithm implements parse recognition without semantic values; we had to add semantic values to the algorithm, but this was a straightforward modification. The overall structure of the algorithm remained unchanged.

PEG. We have a Parsing Expression Grammar [5] interpretation of Gil. It supports all the features of Gil, but interprets choice as deterministic and prioritized (first match). The coroutine translation is fully compatible with the PEG back end, despite the different choice semantics, and required no modifications to be retargeted.

GLR. dypgen [13] is a GLR parser generator written in OCaml. dypgen has native support for "flowing" a value through a parse and supports most of the features of Gil[3], so no modifications to dypgen were necessary to support coroutines. However, implementing the semantics of Gil on dypgen did require a fairly deep and precise understanding of dypgen's semantics.

Memoizing Parser Combinators. Our final back end is a set of parser combinators based on Johnson's memoizing, top-down parser combinators for all context-free grammars [11]. Our combinators are fairly faithful to Johnson's originals, fixing one significant performance problem and adjusting for the differences between Scheme and OCaml. The added support for Gil's context-sensitive features had a trivial impact on the difficulty of implementing the combinators. As with the other back ends, the use of coroutines with parser combinators required no modifications to the coroutine generator.

Although every parsing algorithm is different, our experience with these four back ends convinces us that extending existing context-free parsing algorithms to support the additional features of Gil is usually straightforward, and certainly simpler than extending them with environments, as in our earlier work.

We found that to support Gil, the parsing engine needs three essential elements: (1) it must thread semantic values along with parses; (2) it must include a mechanism for abandoning a parse; and (3) it must support nonterminals parameterized by semantic values. Note that (1) is usually already a feature of any practical parsing tool. For (2), note that we are starting from parsing engines which can already parse all context-free languages, including ambiguous languages; such parsers necessarily include machinery for attempting multiple parses and abandoning failed parses. Parameters for nonterminals (3) are very

[3] Note that dypgen also has support for Gul-style dependency, implemented, in part, with explicit environment manipulations. Nevertheless, our ability to target dypgen based only on the Gil-relevant features demonstrates the applicability of our coroutine translation to a GLR engine.

naturally supported by top-down parsers. In bottom-up parsers they are less common, but we did not find them difficult to implement with dypgen's existing features, for example.

Implementing the coroutine transformation. We use the coroutine transformation in the front end of our Yakker parser generator. While our experience is subjective, we found the translation from paper to software to be straightforward. The core of our actual implementation is nearly identical to the pipeline presented in this paper. The major difference is that our front end supports many features that we have not mentioned, and, consequently, we implemented the coroutine transformation without normalization and erasing, at the cost of additional cases to consider in the other stages of the transformation. Based on our experience, we feel that the formal presentation in this paper contains sufficient information for a practical implementation.

Use cases. Finally, we have written a variety of examples in Yakker's Gul-style language. These examples demonstrate the practical utility of the features supported by Gul, lending weight to the argument for dependent parsing. In addition, our ability to generate working parsers from our grammars demonstrates that our technique works in practice, not just in theory. The examples are described in [9], and include languages like OCaml, JavaScript, Python, and Aurochs PEGs, and data formats including IMAP messages, IETF RFC grammars, Mail.app mailboxes, and the many formats expressible in the PADS languages.

8 Related Work

In earlier work, we presented a formalism that incorporated support for dependent parsing, scannerless parsing, full context-free grammars and foreign parsers [10]. That work focused on the correctness of the translation from grammars to transducers and their execution using an Earley-style algorithm. This work represents a significant advance beyond our previous work. It proposes a fundamentally different, and more general, approach to the handling of dependency, both in theory and in practice. Our separation of binding concerns into a coroutine means that correctness proofs of other techniques can be free of all the binding and environment concerns which played such a prominent role in our previous work. The same benefits carry over to the implementation, as we discussed in Section 7. In addition, our user-level language Gul differs from the grammar language of our previous work in a number of useful ways. It adds lexical scoping of variables, return values for nonterminals, binding to nested right-sides, and a functional interpretation of binding. Also, boxes now have access to the entire input. Moreover, our theory makes explicit the requirements we place on the target language, rather than supposing an untyped lambda calculus.

There are many alternative grammar formalisms for supporting some degree of context sensitivity. We have compared many of the closely related formalisms with our dependent grammars in earlier work [10]. However, we add here a brief

comparison with definite clause grammars (DCGs) [15], a popular formalism for specifying grammars as logic programs. While the power of a particular formulation of DCGs depends on that of the underlying logic language, there are certainly examples with as much power as Gul. Moreover, the support for unification in logic languages provides a more flexible interpretation of variable bindings. However, DCGs rely critically on the features and semantics of the logic language in which they are embedded. In contrast, we are striving to provide an approach to implementing dependent parsing that is compatible with many different parsing algorithms, and applicable across many different programming languages.

Of the existing approaches to handling dependent parsing, the most straightforward is to compile grammars into recursive-descent parsers where binding and expressions in the grammar are copied directly into the target language of the parser. This approach is taken, for example, by the compiler of the PADS data description language [4]. In the case of embedded grammar languages, like monadic parser combinators, the binding support of the target language is even used directly [8,12].

However, this higher-order approach results in much of the grammar being trapped under lambdas, thereby prohibiting useful analyses and transformations which can be critical for performance or even termination (for example, the recent left-corner transform for typed grammars [1]) [12,19]. Paterson proposed the *arrow notation* as an alternative approach that allows such analyses and transformations for arrow combinators [14]. In essence, the translation of the arrow notation to (point-free) arrow combinators "pushes down" binders to the computations that use them, while "lifting" the other elements (in our case, grammar constructs) out from beneath the binders. All bound variables are collected in a tuple which is threaded through the computation.

Our coroutine transformation can be viewed as an alternative translation to point-free style that is better suited to our goal of supporting a wide variety of target languages and reusing existing parsing engines. In particular, we want to support table-based parsers (such as LR, GLR, and Earley parsers) and ML-style typed languages. These parsers require a uniform type for semantic values. Therefore, if we used environment tuples as semantic values, we would have to wrap them in a union type, e.g., to place them on a semantic value stack. Specifying such a union type would require either that the parser generator discover the types of all bound variables or that the user write them down, both of which we want to avoid; the former, so as not to further constrain the choice of target language, and the latter, so as not to burden the user. In our method, the bound values are stored in closures and hence their type is hidden from the type of our semantic values (the coroutines).

Coroutines and parsing have a long history together. For example, Conway originally introduced coroutines as a way to structure a one-pass compiler, including lexer and parser [2], and Warren used them in evaluating the attributes of an attribute grammar [20]. Our work differs from most uses of coroutines in an essential technical detail. In the standard approach, there is dynamically only

one instance of any given coroutine, and, at each invocation, it resumes from the last point at which it yielded. Furthermore, the coroutine itself is responsible for maintaining its state (including its last code location). In contrast, our coroutines are pure (assuming pure embedded actions), which has two major implications: the parsing engine is responsible for maintaining the current version of any coroutine, and it is free to duplicate a coroutine as necessary for exploring different parsing branches.

Technically, our coroutines are closer to trampolined computations than to classic coroutines. Our method of implementing our coroutines is very close to the trampolined style of Ganz, et al., in which a computation is written such that its control flow can be managed externally by a so-called *trampoline* [6]. Our approach is different in two ways. First, because of our concern for sharing nonterminal parses, we needed to extend their method with a novel treatment of call and return. Second, our trampoline—the parsing engine—has a closer relationship with the coroutine. Instead of simply "bouncing" the coroutine at each step, it guides the control flow with the integer argument to the coroutine's closure. Moreover, our coroutines communicate information back to the parsing engine, whether for foreign parsers or for parsing constraints.

References

1. Baars, A., Swierstra, S.D., Viera, M.: Typed transformations of typed grammars: The left corner transform. In: Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications, LDTA 2009, pp. 8–33 (March 2009)
2. Conway, M.E.: Design of a separable transition-diagram compiler. Commun. ACM 6(7), 396–408 (1963)
3. Danvy, O., Filinski, A.: Representing Control: A study of the CPS transformation. Mathematical Structures in Computer Science 2(4), 361–391 (1992)
4. Fisher, K., Gruber, R.: PADS: A domain specific language for processing ad hoc data. In: PLDI 2005: Programming Language Design and Implementation, pp. 295–304. ACM Press, New York (2005)
5. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: POPL 2004: ACM Symposium on Principles of Programming Languages, pp. 111–122. ACM Press, New York (2004)
6. Ganz, S.E., Friedman, D.P., Wand, M.: Trampolined style. In: ICFP 1999: Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming, pp. 18–27. ACM, New York (1999)
7. HTML5 (including next generation additions still in development) Draft Standard October 6 (2010), `http://www.whatwg.org/specs/web-apps/current-work/`
8. Hutton, G., Meijer, E.: Monadic parsing in Haskell. Journal of Functional Programming 8(4), 437–444 (1998)
9. Jim, T., Mandelbaum, Y.: A new method for dependent parsing. Technical Report TD:100334, AT&T Labs—Research (2011)
10. Jim, T., Mandelbaum, Y., Walker, D.: Semantics and algorithms for data-dependent grammars. In: POPL 2010: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 417–430. ACM, New York (2010)

11. Johnson, M.: Memoization in top-down parsing. Computational Linguistics 21(3), 405–417 (1995)
12. Leijen, D., Meijer, E.: Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht (2001)
13. Onzon, E.: Dypgen: self-extensible parsers and lexers for OCaml, `http://dypgen.free.fr/`
14. Paterson, R.: A new notation for arrows. In: ICFP 2001: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming, pp. 229–240. ACM, New York (2001)
15. Pereira, F.C.N., Warren, D.H.D.: Definite clause grammars for language analysis. Artificial Intelligence 13, 231–278 (1980)
16. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
17. Scott, E.: SPPF-style parsing from Earley recognisers. In: Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007). Electronic Notes in Theoretical Computer Science, vol. 203, pp. 53–67. Elsevier, Amsterdam (2008)
18. Scott, E., Johnstone, A.: Right nulled GLR parsers. ACM Transactions on Programming Languages and Systems 28(4), 577–618 (2006)
19. Swierstra, S.D., Azero Alcocer, P.R.: Fast, error correcting parser combinatiors: A short tutorial. In: Bartosek, M., Tel, G., Pavelka, J. (eds.) SOFSEM 1999. LNCS, vol. 1725, pp. 112–131. Springer, Heidelberg (1999)
20. Warren, S.K.: The coroutine model of attribute grammar evaluation. PhD thesis, Rice University, Houston, TX, USA (1976)