

The Structure of Typed Programming Languages

Chapter 5 and Beyond

CS386L- Programming Languages

Dr. Greg Lavender

Department of Computer Sciences

The University of Texas at Austin

Chapter 5

- Records and Lambda abstractions
 - we can desugar the CIL and reconstruct it just using records and lambda abstractions
 - use fewer syntactic domains
 - use type rules to enforce proper syntax
 - the semantics stays pretty much the same
 - are the two key elements of the “principles” that we have studied
 - abstraction principle
 - parameterization & correspondence principles
 - qualification principle
 - records and lambda abstractions coexist without interfering with one another

Desugared Language

- three domains and one syntax rule

$E \varepsilon$ Everything

$N \varepsilon$ Numeral

$L \varepsilon$ Location

$$E ::= E_1 := E_2 \mid E_1 ; E_2 \mid \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \text{ fi} \mid \\ \text{skip} \mid \text{while } E_1 \text{ do } E_2 \text{ od} \mid E_1 + E_2 \mid E_1 = E_2 \mid \\ \sim E \mid @E \mid N \mid L$$

- type rules distinguish commands from expressions, locations and numbers!
 - rules enforce well-typed programs

Record Introduction

- reapply the abstraction and qualification principles

$E ::= \dots \mid I=E \mid E_1, E_2 \mid \text{with } E_1 \text{ do } E_2 \mid I$

- $I=E$ is a name binding, called a "record"
 - like $\{I=E\}$ syntax from before
- E_1, E_2 joins two records
 - $(A=2, B=\text{skip}), C=(A=@\text{loc}_1+1) \Rightarrow A=2, B=\text{skip}, C=(A=@\text{loc}_1+1)$
 - $\text{with } E_1 \text{ do } E_2$ is a desugared $\text{begin } E_1 \text{ in } E_2 \text{ end}$
- records allow all phrases from all syntax domains to be named - the abstraction principle
- the with construct allows all phrases to admit local definitions - the qualification principle

Lambda Abstraction Introduction

- reapply the parameterization principle
 - $E ::= \dots \mid \lambda I:\theta.E \mid E_1 E_2 \mid I$
- this is a desugared version of a parameterized binding
 - define $I_1(I_2:\theta)=E$ becomes $\lambda I:\theta.E$
 - lambda abstraction
 - invoke $I_1(V)$ becomes $E_1 E_2$
 - like application in the lambda calculus
 - note that E_1 can be any expression, for example, an expression that evaluates to a function to which E_2 is applied
 - in Scheme: `((if (< a b) + *) 2 3)`
 - I is just a parameter reference
- lambda abstraction allows phrases from any syntax domain to be parameters
 - the parameterization principle

The Redefined Language

- See figure 5.2
 - full syntax
 - all types rules
 - this modified imperative language admits well formed programs than the previous one
 - does it look familiar?

Orthogonality

- orthogonality of features is a highly desired feature of a language
 - a feature is orthogonal to another feature if they are independent and can coexist without the other
 - when adding a new feature to a language, you can determine whether or not it is orthogonal if
 - existing semantics are not changed by the new feature, but perhaps extended
 - if compound phrases are used by the new feature, then it applies uniformly, and does not require a lot of special case definitions
 - records extend semantics by adding the environment argument, as we already saw
 - lambda abstractions are also orthogonal
 - can you think of a non-orthogonal feature? what about newint?

Domain Specific Languages

- An important application of the concept of orthogonality is in DSLs
 - consider defining a “base” language, not unlike the core imperative language we have studied
 - then provide a “toolkit” for extending the language to have domain-specific features
 - need to allow for syntactic extension
 - either via the grammar or via a macro feature
 - need to allow for the addition of new type rules
 - and the type system that enforces them
 - need to allow for the extension of the underlying interpreter/compiler
 - since this is where the semantics is implemented
 - build an “object-oriented” interpreter that provides extensible interfaces so that the semantics can be extended/modified as needed since there will no doubt be features that are not perfectly orthogonal, and so it may become necessary to modify the semantics
 - the interpreter should also include “assertions” that ensure that the semantics are not violated as a result of an extension or local modification required for some new feature

What Next?

- highly recommend chapters 8 & 9
 - Higher-Order Typed Lambda Calculi
 - Propositional Logic Typing
 - basic discussion of the Curry-Howard Isomorphism
- See separate handout for a more comprehensive and advanced reading list
 - if there were enough interest, I might be willing to conduct an advanced topics seminar on these topics in the Fall term

Future Topics of Study

- **Gentzen's Natural Deduction**
 - and the sequent calculus
- **Typed Lambda Calculi**
 - and related type theories
- **Intuitionistic logic and type theory**
 - Brouwer's intuitionism
 - Martin-Lof's intuitionistic type theory
- **Curry-Howard Isomorphism**
 - propositions as types
 - proofs as programs
- **Category theory for computer science**
 - cartesian closed categories
 - toposes (topoi)
 - higher-order categorical logic

Gentzen's Natural Deduction

- Gerhard Gentzen, 1909-1945
 - added a list of assumptions (i.e., given propositions) to logical judgments to make deductions more "natural"
 - $B_1, \dots, B_n \vdash A$
 - Γ and Δ are often used to stand for lists of assumptions
 - Γ, Δ is the union of two lists of assumptions
 - if the assumptions B_i are all empty, then the logic is equivalent to Frege's
 - introduction (I) and elimination (E) rules

$$\frac{}{A \vdash A} \text{Id}$$

$$\frac{\Gamma, B \vdash A}{\Gamma \vdash B \rightarrow A} \rightarrow I$$

$$\frac{\Gamma \vdash B \rightarrow A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A} \rightarrow E$$

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \& B} \&I$$

$$\frac{\Gamma \vdash A \& B}{\Gamma \vdash A} \&E_1$$

$$\frac{\Gamma \vdash A \& B}{\Gamma \vdash B} \&E_2$$

Typed Lambda Calculus

- Church introduces the typed lambda calculus in 1940
 - goal was to avoid the paradoxes of logic
 - Russell introduce ramified types for this purpose in set theory
 - it an amazing fact that the entire theory of computing (as term rewriting) can be characterized with just three terms and one reduction rule
 - augmented with the Y combinator for recursive definitions
- types
 - $A \rightarrow B$ is the type of a function from type A to B
 - $A \& B$ is the type of a pair (A, B)
 - $x_1:B_1, \dots, x_n:B_n \vdash t:A$ is a typing judgment
 - t has type A if each x_i has type B_i

Typed Lambda Calculus

- function application
 - use Γ and Δ for lists of var:type pairs
 - type rule
 - premises
 - t is a function from type B to type A under assumption Γ
 - u is a term of type B under assumption Δ
 - conclusion
 - $t(u)$ is a term of type A under the combined assumptions Γ, Δ

$$\frac{\Gamma \vdash t:B \rightarrow A \quad \Delta \vdash u:B}{\Gamma, \Delta \vdash t(u):A}$$

Type Lambda Calculus Rules

- expressed in a natural deduction form
 - note the structural similarity with Gentzen's rules
 - one rule for terms that are variables
 - rules for functions or pair types
 - introduction and elimination rules
 - lambda abstraction introduces a function
 - application eliminates a function
 - a pair constructor creates a pair, a selector eliminates it

$$\begin{array}{c}
 \text{----- Id} \\
 x:A \vdash x:A
 \end{array}
 \qquad
 \begin{array}{c}
 \Gamma, x:B \vdash t:A \\
 \text{----- } \rightarrow I \\
 \Gamma \vdash \lambda x.t:B \rightarrow A
 \end{array}
 \qquad
 \begin{array}{c}
 \Gamma \vdash t:B \rightarrow A \quad \Delta \vdash u:B \\
 \text{----- } \rightarrow E \\
 \Gamma, \Delta \vdash t(u):A
 \end{array}$$

$$\begin{array}{c}
 \Gamma \vdash t:A \quad \Delta \vdash u:B \\
 \text{----- } \&I \\
 \Gamma, \Delta \vdash \langle t, u \rangle : A \& B
 \end{array}
 \qquad
 \begin{array}{c}
 \Gamma \vdash t:A \& B \\
 \text{----- } \&E_1 \\
 \Gamma \vdash t.fst:A
 \end{array}
 \qquad
 \begin{array}{c}
 \Gamma \vdash t:A \& B \\
 \text{----- } \&E_2 \\
 \Gamma \vdash t.snd:B
 \end{array}$$

Typed Lambda Calculus

- beta reduction preserves types of terms
 - reduction of pairs always “shrinks” a derivation tree
 - reduction of a lambda abstraction may increase the size of the tree, since substitution can occur multiple times in the body of the abstraction
 - however the types are kept simple since $\lambda x.E:B \rightarrow A$ reduces to type A
 - this is good since the Church-Rosser property says that if there is a normal form, there is only one, so we eventually reach one type for the normal form

Type Systems

- many type systems have been designed based on the typed lambda calculus
 - sum types
 - record types
 - abstract types
 - type classes
 - monad types
 - linear types (linear logic)
 - etc.
- types provide provide a key language design criteria
 - start from the types and the rest will follow
 - terms follow as introducers and eliminators of types
 - reductions follow when an introducer meets an eliminator
 - type safety follows from showing the each reduction preserves type derivations according to the type rules

Curry-Howard Isomorphism

- Correspondence between Gentzen's natural deduction and typed lambda calculus
 - given a type derivation, you can construct the corresponding proof, and vice versa
 - term reduction corresponds to proof simplification
 - the correspondence extends to cover disjunction
 - $A || B$ corresponds to disjoint union (variant records)
 - correspondence works best when applied to intuitionistic logic
 - denies the law of the excluded middle $A || !A$
 - but has been shown to work with classical logic

Final Reading Assignment

- Highlights of the History of the Lambda Calculus, by J. Barkley Rosser
 - "Kleene-ness is next to Godel-ness"
- Proofs are Programs: 19th Century Logic and 21st Century Computing, by Philip Wadler
 - simply illustrates the curry-howard isomorphism by showing that Gentzen's natural deduction rules for propositional logic correspond to functions in the typed lambda calculus and propositions correspond to types
 - hence, the proofs-as-programs and propositions-as-types correspondences first noted by Curry and expanded/published by Howard
 - but this is not really an isomorphism between logic and programming, since there are MANY other real programming constructs that do not fit into this correspondence, but it is close to functional programming