Type-Driven Language Design

Luca Cardelli

Digital Equipment Corporation

Systems Research Center

PLDI'95 Tutorial, La Jolla, June 18

WITH COMMENTS

Abstract

Over the decades, the focus of program development has shifted from the design of algorithms to the design of structures. Progress has been marked by the introduction of new structuring mechanisms, culminating (for now) in concepts such as data abstraction, objects, and modules. The overall goal has been to make program components increasingly reusable.

A similar evolutionary path can be traced for programming languages: from the early algorithmic languages, to increasingly data-centric and object-centric ones. The overall technique has been the introduction of new type structures: advances in program structuring were progressively embedded into type structures; conversely, new type structures enhanced our ability to structure programs.

As an interesting result of this feedback loop, language features have become clustered according to types, so that the type structure largely determines the flavor of the language. These clusters of features have become increasingly modular and “reusable” from one language to the next.

It seems natural to turn this trend into a conscious goal. Type-oriented clustering of features is an effective technique for language analysis and design; one that emphasizes orthogonality. It can be used both to understand existing (deficient) languages, and to produce new (wonderful) ones. Show me your types, and I'll show you your language.

Introduction

- Theme: how type structures have influenced and are influencing language design.

- Since the days of structured programming, language design has focused on *type structures*: on data structures and program organization, not on control structures. Programming languages with rich type structures, like Pascal, emerged.
- Interestingly, structured programming aimed at first to regiment *control structures*: this already indicates some deep connection between control structures and type structures. (This connection will be discussed in detail.)

- The design difficulties encountered since those days are causing us to focus even more deliberately on type structures.
- I do not prescribe a complete solution to the problem of designing languages. I describe how the remaining hard problems in language design are being attacked right now. We believe this direction is promising.

Three parts: Past history and trends (25min), current knowledge and problems (25min), future trends and approaches (40min).

A Brief History of Type

- Typing was first introduced for better code generation and run-time efficiency.
  - *Economy of execution.*
- Then typing became useful for separate compilation via interfaces and modules.
  - *Economy of compilation.*
- Soon it became clear that typing had methodological advantages.
  - *Economy of development.*
- Thus typing became a major concern in language design, particularly for reasons of orthogonality.
  - *Economy of language features.*

  Regular, orthogonal type structures eliminate special cases, made languages simpler.
- Only recently, conscious efforts have been made to let formal type theory drive language design. Even ML was mostly inspired by denotational semantics.

Progression of Concerns

Why are type structures interesting?

- What we want to do:
  - 1) Program design and implementation.

    From the small task of programming...
  - 2) Application design and implementation.

    To the larger task of writing applications made up of programs...
  - 3) Language design and implementation.

    To the grandious task of desgning language for writing applications.

For all these activities we need good structuring principles:

Structuring Principles == Type Structures

Type structures are (by definition) structuring principles that can be *checked*.

To perform these activities we need the following.

- We need:
  - 1) Data structure types for program construction.

    For constructing programs.
  - 2) Object/module types for application construction.

    For constructing applications.
  - 3) Coherent (and sound) type systems for language construction.

    For constructing languages.

- We claim that type structures are good for:
  - 1) Program design. To write clean, maintainable algorithms.
  - 2) Application design. To write clean modular applications.
  - 3) Programming language design. To design clean orthogonal languages.

The first two claims are well-known and widely discussed. We concentrate on the third one, and on how it follows from the other two.

Type Structures

What are type structures?

Many programming langauges make widespread use of *static (type) information*, intended as a (*partial) specification* of a program.

With respect to general specs:

- Type structures should be (largely) *decidably verifiable*; the purpose of typing constraints is not simply to state programmer intentions, but to actively trap programmers errors. (Arbitrary formal specifications do not have this property.)
- Type structures should be *transparent*: a programmer should be able to predict easily when a program will typecheck, and if it fails to typecheck the reason should be clear. (Automatic theorem proving does not have this property.)
- Type structures should be *enforceable*: statically checked as much as possible, otherwise dynamically checked. (Program comments and conventions do not have this property.)

Diverse language paradigms, imperative, functional, concurrent, object-oriented, etc. have been converging towards a common methodology.

Languages for describing large and well-structured software systems have been evolving towards stronger and stronger type structures.

(Ordered by “sophistication” of type system: not chronology)

Algorithmic:	Fortran → Algol-60 → Pascal → CLU, Modula-2
System:	Assm → C → Modula-3, Oberon, C++
Object-oriented:	Smalltalk → Simula67 → Modula-3, Eiffel, C++
Functional:	Lisp → Scheme → ML, Haskell
Concurrent:	Semaphores → Monitors → NIL, FX
Logical:	Prolog → ?

There have been also serious attempts to typecheck untyped languages such as Smalltalk, Lisp/Scheme, and Prolog.

We can separate this evolutionary process into two phases: the easy part, and the hard part.

The Easy Part

In the beginning, there were “monomorphic” type structures:

The early days

- Integers and floats (occasionally, also booleans and voids).
- Monomorphic arrays (Fortran).
- Monomorphic trees (Lisp).

The days of structured programming

- Product types (records in Pascal, structs in C).
- Union types (variant records in Pascal, unions in C).
- Function/procedure types (often with various restrictions).
- Recursive types (typically via pointers).

End of the easy part

This phase culminated with user-definable monomorphic types obtained by combining the constructions above (Pascal, Algol68).

We learned how to design reasonable languages. Nowadays, any good student should be able to combine these ingredients into a not-too-broken language design.

Still, people who ignore history are doomed to repeat it...

The Hard Part

Four major innovations

- Objects and Subtyping (Simula 67).
- Abstract types (CLU).
- Polymorphism (ML).
- Modules (Modula 2).

Despite much progress, nobody really knows yet how to combine all these ingredients into coherent language designs.

Confusion

These four innovations are partially overlapping and certainly interact in interesting ways. It is not clear which ones should be taken as more prominent. E.g.:

- Object-oriented languages have tried to incorporate type abstraction, polymorphism, and modularization all at once. As a result, o-o languages are (generally) a mess. Much effort has been dedicated to separating these notions back again.
- Claims have been made (at least initially) that objects can be subsumed by either higher-order functions and polymorphism (ML camp), by data abstraction (CLU camp), or by modularization (ADA camp).
- One hard fact is that full-blown polymorphism can subsume data abstraction. But this kind of polymorphism is more general than, e.g., ML's, and it is not yet clear how to handle it in practice.
- Modules can be used to obtain some form of polymorphism and data abstraction (ADA generics, C++ templates) (Modula 2 opaque types), but not in full generality.

So, how to we obtain a language that has a good integration of objects, abstractions, polymorphism, modules, or a subset of those?

Language Design (Still?)

Is there any language design left to do?

Certainly yes within new, specialized areas.

But even within “classical” programming languages, a perfect integration of those four innovations has not been achieved.

Is there a principled path to solutions?

Let’s consider again the evolution of programming.

A Shift in Programming

- Programs were initially seen as consisting mostly of algorithms.
- Gradually, the emphasis shifted to the description of data structures and module structures, with algorithms seen as “attached” to such structures.
- This trend was embodied into data abstraction, object-orientation, and modularization methodologies: from procedure-centric to data-centric, object-centric, and module-centric.

A Parallel Shift in Languages

- Languages were initially seen as a collection of control structures for expressing algorithms; e.g. the various forms of DO, FOR, and WHILE loops.
- Gradually, the emphasis shifted to the description of data types, object types (classes) and module types (interfaces), with control structures “attached” to such new constructions.
- Eventually, languages were designed specifically to embody the developing data abstraction, object-orientation, and modularization methodologies.
- Thus we moved from algorithmic languages to “bureaucratic” languages. (Bureaucracy: set up the structure before you write any code; fix the data before you write any algorithms.)

The Design Trend

- Progressive introduction of new type structures.
  “type structures”:
  - Statically checked structures that have little or no effect on execution.
  - Structures that intentionally “get in the way” of writing (undisciplined) code.
- A feedback loop:
  - Spontaneous structuring trends (“methodologies”) were embedded as type structures within new language designs.
  - Vice-versa, new language designs increased the ability to structure programs, and generated new methodologies.

- Effect of the feedback loop:
  - Language features have coagulated around type structures.
  - These clusters of types and features can be largely “reused” across languages.
  - This is the closest we ever came to modular language design.
- A path for the future
  - The easy language features have been designed already. Because of the difficulty of the remaining problems, we had better go about it in well-organized way.
  - We should search for modular mechanism organized around type structures.

END OF PART I. What we learned: (1) a trend towards increasing structuring; (2) a hypothesis to extend the trend to future designs.

ORGANIZING FEATURES BY TYPE

- There is a formal correspondence between constructive logic (a.k.a. "type theory" in the broadest sense) and programming. This is because constructive logic is a logic of (functional) computation. The correspondence holds, at least informally, even for imperative computation.
- Already, constructs studied in type theory have provided new insights on constructs independently developed in programming (e.g. for abstract types and polymorphism).
- This kind of theoretical background is very useful to understand, simplify and generalize programming constructs. Since the constructs studied in type theory are more general that those normally used in programming, there is still scope for expansion and, hopefully, for solving open language design problems.
- (However we should not oversimplify: a programming language is not just a formal system. It has very different requirements and constraints in terms of compiler engineering, user aesthetics, and programming sociology. Type theory is a metaphor.)

- Formal type theories are presented as orthogonal collections of type operator and related constructions. Type rules are classified according to a standard formation-introduction-elimination pattern.
- This leads to modular formal systems. If applied to language design, it leads to modular language designs.
- In an evolutionary sense, a “successful feature” is one that is preserved from one language to the next, and therefore must be relatively modular.

Why Types? (a methodological view)

To aid in the evolution of software systems:

- Large software systems are not created, they evolve.
- Evolving software systems are (unfortunately):

  not correct

  (people keep finding and fixing bugs)

  or else, not good enough

  (people keep improving on space and time requirements)

  or else, not clean enough

  (people keep restructuring for future evolution)

  or else, not functional enough

  (people keep adding and integrating new features)

- Some form of “software hygiene” is necessary.

Reliability

Naively, software either works or it does not, but evolving software is always sort of in-between.

Working hardware is reliable if it does not break too often, in spite of wear.

Evolving software is reliable if it does not break too often, in spite of change.

Type systems provide a way of controlling change, inspire some degree of confidence after each evolutionary step, and help in producing and maintaining reliable software systems.

Introduction and Elimination Constructs

Most language constructs can be classified into *introduction* constructs and *elimination* constructs.

Consider the following simple untyped language:

		Introduction	Elimination (may fail)
variables	x		
constants		**true**, **false**	**if** a **then** b **else** c **end**
functions		**fun**(x) b	b(a)
records		**record** l_1=a_1, ... , l_n=a_n **end**	b.l

This language is flexible, because untyped, but computations may fail.

Failure points reduce software reliability.

To prevent failure, we organize terms into a type system.

- Monomorphic type systems can eliminate failure points:

```
if (fun(x) x) then ... else ... end
(record end) (false)
true.l
```

- More sophisticated, polymorphic, type systems try to preserve the flexibility of the untyped calculus as much as possible:
  - Preserved by parametric polymorphism:

```
(fun(x) record fst=x snd=x end) (fun(y)y)
(fun(x) record fst=x snd=x end) (true)
```

  - Preserved by subtype polymorphism:

```
(fun(x) x.t) (record t=true end)
(fun(x) x.t) (record t=true, u=false end)
```

So, how do type systems work, and how do we express them?

The Language of Type Theory

Type structures are expressed via a simple and flexible formalism.

Environments ≡ Interfaces

$$E \equiv \text{Animal} :: \text{Type}, \text{Dog} <: \text{Animal}, d : \text{Dog}$$

Judgments ≡ Assertions

$E \vdash$

$E \vdash \text{Animal}$

$E \vdash d : \text{Animal}$

Rules Assumptions => Conclusions

$$\frac{}{\emptyset \vdash} \qquad \frac{E \vdash c : \mathrm{Cat} \qquad E \vdash d : \mathrm{Dog}}{E \vdash c \,\&\, d : \mathrm{Fight}}$$

$$\frac{E \vdash}{E \vdash \mathrm{spot} : \mathrm{Cat}} \qquad \frac{E \vdash}{E \vdash \mathrm{fido} : \mathrm{Dog}}$$

Derivations

$$\frac{\dfrac{\dfrac{}{\emptyset \vdash}}{\emptyset \vdash \mathrm{spot} : \mathrm{Cat}} \qquad \dfrac{\dfrac{}{\emptyset \vdash}}{\emptyset \vdash \mathrm{fido} : \mathrm{Dog}}}{\emptyset \vdash \mathrm{spot} \,\&\, \mathrm{fido} : \mathrm{Fight}}$$

Type errors

Absence of derivations. ($\emptyset \vdash$ fido : Cat)

The formation-introduction-elimination pattern

Rules are organized in a certain style; not a law of nature, but a useful classification.

Formation rules

What a type is. (Structure.)

Introduction rules

How the elements of a type are created. (Allocation.)

Elimination rules

How the elements of a type are used. (Control.)

(Reduction rules)

How the elements of a type are evaluated. (Computation.)

Abstract Example: Cartesian Product Types

Formation

A product of two types is a type.

(× Formation)

$$\frac{E \vdash A \qquad E \vdash B}{E \vdash A \times B}$$

Introduction Introduces ×

A pair of terms has a product type, provided the terms have the corresponding factor types.

(× Introduction)

$$\frac{E \vdash a : A \qquad E \vdash b : B}{E \vdash \langle a,b \rangle : A \times B}$$

Elimination (with) Eliminates ×

If a term has a product type, the type of its first component is the first factor.

If a term has a product type, the type of its second component is the second factor.

(× Elimination fst)

$$\frac{E \vdash c : A \times B}{E \vdash \mathrm{fst}(c) : A}$$

(× Elimination snd)

$$\frac{E \vdash c : A \times B}{E \vdash \mathrm{snd}(c) : B}$$

Multiple elimination rules can usually be combined into a single rule for a single control structure. In this case:

(× Elimination)

$$\frac{E \vdash c : A \times B \qquad E, x{:}A, y{:}B \vdash d : D}{E \vdash \mathbf{with}\ \langle x,y \rangle = c\ \mathbf{do}\ d\ \mathbf{end} : D}$$

This is a CONTROL STRUCTURE that arises naturally from a data structure.

Record types

Formation

What is a good record type.

Introduction

How to construct a well-typed record.

Elimination (with)

Dot notation, or Pascal’s **with**.

Union types

Formation

(+ Formation)

$$\frac{E \vdash A \qquad E \vdash B}{E \vdash A + B}$$

Introduction

(+ Introduction left)

$$\frac{E \vdash a : A}{E \vdash \mathbf{inl}(a) : A + B}$$

(+ Introduction right)

$$\frac{E \vdash b : B}{E \vdash \mathbf{inr}(b) : A + B}$$

Elimination (case)

(+ Elimination)

$$\frac{E \vdash c : A + B \qquad E, x{:}A \vdash a : D \qquad E, y{:}B \vdash b : D}{E \vdash \mathbf{case}\ c\ \mathbf{of}\ \mathbf{inl}(x)\ a \ \mathbf{|}\ \mathbf{inr}(y)\ b\ \mathbf{end} : D}$$

Procedure types

Formation

(→ Formation)

$$\frac{E \vdash A \qquad E \vdash B}{E \vdash A \rightarrow B}$$

Introduction

(→ Introduction)

$$\frac{E, x{:}A \vdash b : B}{E \vdash \lambda(x{:}A)b : A \rightarrow B}$$

Elimination (call)

(→ Elimination)

$$\frac{E \vdash c : A \rightarrow B \qquad E \vdash a : A}{E \vdash c(a) : B}$$

Boolean types

Formation

(Bool Formation)

$$\frac{E \vdash}{E \vdash \mathrm{Bool}}$$

Introduction

(Bool Introduction true)

$$\frac{E \vdash}{E \vdash \mathbf{true} : \mathrm{Bool}}$$

(Bool Introduction false)

$$\frac{E \vdash}{E \vdash \mathbf{false} : \mathrm{Bool}}$$

Elimination (conditional)

(Bool Elimination)

$$\frac{E \vdash a : \mathrm{Bool} \qquad E \vdash b : D \qquad E \vdash c : D}{E \vdash \mathbf{if}\ a\ \mathbf{then}\ b\ \mathbf{else}\ c\ \mathbf{end} : D}$$

Numeric types

Formation

(Nat Formation)

$$\frac{E \vdash}{E \vdash \mathrm{Nat}}$$

Introduction

(Nat Introduction **0**)

$$\frac{E \vdash}{E \vdash \mathbf{0} : \mathrm{Nat}}$$

(Nat Introduction **succ**)

$$\frac{E \vdash n : \mathrm{Nat}}{E \vdash \mathbf{succ}(n) : \mathrm{Nat}}$$

Elimination (primitive recursion)

(Nat Elimination)

$$\frac{E \vdash n : \mathrm{Nat} \qquad E \vdash b : D \qquad E, i{:}\mathrm{Nat}, x{:}D \vdash c : D}{E \vdash \mathbf{for}\ x = b, i < n\ \mathbf{do}\ c\ \mathbf{end} : D}$$

Array types

Formation

(Array Formation)

$$\frac{E \vdash A}{E \vdash \mathrm{Array}(A)}$$

Introduction

(Array Introduction)

$$\frac{E \vdash a_1 : A \quad \ldots \quad E \vdash a_n : A}{E \vdash \mathrm{array}(a_1, \ldots, a_n) : \mathrm{Array}(A)}$$

Elimination (iteration)

(Array Elimination)

$$\frac{E \vdash a : \mathrm{Array}(A) \qquad E, i : A \vdash b : \mathrm{Void}}{E \vdash \textbf{for}\ i\ \textbf{in}\ a\ \textbf{do}\ b\ \textbf{end} : \mathrm{Void}}$$

Less Trivial Examples

- Recursive types
  - Fold/unfold; structural recursion.
- Algebraic types
  - Elimination: pattern matching.
- Exception types
  - Elimination: raise.
- Continuation types
  - Elimination: call/cc; goto.
- Dynamic types
  - Elimination: typecase.

More importantly, what about the four major innovations? For these, our systematic pattern should really begin to pay off, because we are getting into more complicated constructions. END OF PART II. What we learned: (1) type structuring organizes language features in neat little packages.

Object types

We now see how to use the patten to attack the hard constructs. We derive some programming design lessons from the process. No more rules are discussed.

Formation

What is a good object type.

Subtyping We extend the pattern with this new judgment.

When two object types are in subtype relation.

Introduction

How to construct a well-typed object.

Elimination

Method invocation, field selection.

But what about Classes?

- We distinguish:
  - Objects have object types (which are similar to record types).
  - Classes are object generators (which are similar to functions returning objects).
  - Classes must have types too: those are the class types.
- N.B. Typical o-o languages, instead, confuse classes with types.

Language design lessons

- The (rather recent) separation of subtyping from inheritance arises naturally from typing consideration. Inheritance has to do with class types, while subtyping has to do with object types.
- Emphasis on object types leads naturally to systematize object-based (as opposed to class-based) languages.

Abstract types

Formation

What is an abstraction.

Introduction

How to construct an abstraction with a given implementation.

Elimination (open)

How to use an abstraction independently of its implementation.

Language design lessons

- Opaque types and their “gensym” property can be explained rationally. E.g., structural type equality makes perfect sense even for abstract types.
- We can describe precisely the rules that prevent abstractions from being violated.
- We can investigate the subtle differences between abstract types and modules.

Polymorphic types

Formation

What is a polymorphic type.

Introduction

How to construct a polymorphic value (often, a type-parametric function).

Elimination (specialize)

How to specialize a polymorphic value (function).

Language design lessons

- Polymorphism (in full generality) can express type abstraction.
- If a language has both some (limited) forms of polymorphism and type abstraction, we can tell whether they are uniformly defined.

Module types

Formation

What is an interface.

Introduction

How to construct a module that matches an interface.

Elimination (import)

How to import and use an interface, without relying on the module implementation.

Language design lessons

- Interfaces are very similar to abstract types, but need not be “types”. That’s because modules need not be “values”.
- Some issues are more important with modules than with abstract types: transparency, parameterization, phase distinctions (separate compilation).
- Modules are harder than they look. The type systems for modules are hard too.

Non-examples (so far)

- Concurrency.
- Security.

Orthogonal Combinations

Type constructions mix orthogonally, sometimes:

- polymorphism + subtyping (= bounded polymorphism).
- abstract types + subtyping (= semi-opaque types).
- polymorphism + abstract types (= parametric abstractions: Stack(X), etc.)

Harder combinations:

- polymorphism + “data layout”.
- abstract types + typecase.
- modules + abstract types.
- modules + polymorphism.
- subtyping + inheritance (F-bounded; “matching”)

Future Directions

Type theory is guiding / should guide language design in the following hot areas:

Object/Class types

- Designing clean, flexible, and largely statically typed o-o languages.
- Emphasis on object types leads naturally to object-based languages. Classes have types too; thus we get class-based languages. We can also get both at once.

Beyond subtyping

- Flexible o-o parameterization (“matching”). C.f. PolyTOIL, Theta. Advanced type-theoretical techniques directly applied to new language designs.

Module types

- Designers often “forget” to include modules in their languages, and regret it later. If we had a good and reusable understanding of modules, maybe disasters would be avoided.
- Advanced module systems are still in a state of flux. Recent attempts focus around highly sophisticated type theories.

Polymorphism and Data Layout

- How can polymorphic type systems efficiently describe data layout?

Type-Driven Designs

Classified according to the *main question* that a language design poses/posed.

Non-examples

- Fortran (“How do loops work?”)
- Algol60 (“How do static scoping and recursion work?”)
- C (“How is data laid out?”)
- Simula/C++ (“How does method lookup work?”)

Examples

- Pascal (“How do type definitions work?”)
- Modula-2 (“How do interfaces work?”)
- CLU (“How does type abstraction work?”)
- ML (“How does polymorphism work?”)
- Oberon / Modula-3 (“How do object types and subtyping work?”)

Current/future examples

- (Quest) Tool (“How does higher-order polymorphism work?”)
- PolyTOIL (“How does inheritance work?”)
- ML2000 (“Type theory from the ground up!”)

TYPE SOUNDNESS

Suppose we have finally succeeded in designing a language according to the prescriptions of type theory. What's the use? One use is proving typing soundness.

- Today, we have the formal tools to prove type systems sound for practical languages.
- No high-power theory required; just operational semantics and a subject reduction theorem (execution preserves typing).
- Full scale soundness proofs are possible. Shortcuts are also possible, especially given previous knowledge of trouble spots: this may be sufficient to give sufficient confidence.

“Why bother? Type systems are easy!”

- Some common trouble spots:
  - Polymorphism and side-effects/exceptions/continuations (even ML was unsound, at times).
  - Subtyping and method parameters (some people have trouble understanding that contravariance is a *fact*, not an opinion).
  - Subtyping recursive types.
  - Data abstractions that travel over the network (“gensym” techniques only work in one address space).
  - Object identity (acquiring, preserving, removing roles in DBPL’s).
  - Security in web languages (type soundness is a prerequisite).

Steps

Syntax

– Define the grammar of the language: the set of terms.

a,b ::= x | **true**| **false** | **if** a **then** b **else** c **end**
| **fun**(x) b | b(a) | **record** l_1=a_1, ... , l_n=a_n **end** | b.l

– Define the notion of free occurrences of variables in terms.

FV(x) = {x}
FV(**fun**(x) b) = FV(b) - {x}

– Define the notion of substitution.

(**fun**(x) b){y←a} = **fun**(z) (b{x←z}{y←a}) where z $\notin$ FV(b)∪FV(a)

– Define the trivial identifications (e.g. renaming of bound variables).

fun(x) x = **fun**(y) y
record l_1=a_1, l_2=a_2 **end** = **record** l_2=a_2, l_1=a_1 **end**

Typing

- Define the rules for forming environments.

$$\frac{}{\emptyset \vdash} \qquad\qquad \frac{E \vdash A \qquad x \notin \mathrm{dom}(E)}{E, x{:}A \vdash}$$

- Define formation-introduction-elimination rules for all types and constructs.

Semantics

- Decide what is an “error” that violates static typing.

  inc(true)

- Decide what instead is an “exception”.

  1/0

- Define a notion of “results” of evaluation, including errors. (N.B. results need not be terms.)

  integers
  memory locations
  closures

- Define an operational semantics. (Usually, an inductively defined relation mapping terms to results, while modifying a global store.)

  $\sigma, S, 1+2 \leadsto 3, \sigma$

- Prove that the semantics is completely defined: every term reduces to a result (possibly an error result) or diverges, but does not “get stuck”.

(N.B. up to here, this process could be called “how to define a language”.)

Type soundness

- Define what it means for a result to have a type (w.r.t. a store). (N.B. error results are not given types.)
- Prove a subject reduction theorem: show that if a term has a type, and the term reduces to a result, then the result has the same type (hence the result is not an error). Therefore, well-typed terms do not "go wrong".

• For a worked-out example involving a (very) small imperative object-oriented language, see: Abadi, M. and L. Cardelli, **An imperative object calculus: basic typing and soundness**. *Proc. Second ACM SIGPLAN Workshop on State in Programming Languages*, 19-32. Technical Report UIUCDCS-R-95-1900, University of Illinois at Urbana Champaign. 1995.

Complexity and benefits of the task

- Proving soundness is not theoretically hard (any more); it is mostly a matter of proof engineering. But it is still operationally hard.
- Full and formal semantics and type soundness proof for a language can be compared in size and complexity to the task of engineering a compiler for the same language.
- Confidence building:
  - When designing a language, we may think the language is sound because we cannot think of a counter-example right now.
  - When proving soundness, we argue why counter-examples cannot exists, but systematically analyzing all possibilities.
  - Even if the proof is not completely formal, we get a deeper understanding of why the language is sound. This at least increases confidence, by forcing one to think harder.

CONCLUSIONS

- A background in type theory helps identify, or avoid, non-trivial trouble spots in language design. Moreover, it helps in building more orthogonal languages.
- These days, when we design new languages we hardly ever design new control structures. Rather, we concentrate on new ways of organizing programs and data.
- Type theory helps both in design, and also in analisys: since most typing fragments are standardized, the type structure is the best place to start when trying to understand a new language.