Inference-driven resource managemenent and polymorphism in systems programming

Jiří Klepl

MFF UK, September 2022

Introduction

High-level languages and systems programming

General motivating problem:
Coding low-level systems in simple languages is laborous and error-prone.

Desirable language features:

- **Good generics** to prevent repetition
- **Type inference** to reduce redundant annotations
- **Deep typechecking** to prevent errors
- **Resource management** for runtime correctness

✖ **Unnecessary overhead**

High-level languages and systems programming

General motivating problem:
Coding low-level systems in simple languages is laborous and error-prone.

Desirable language features:

- **Good generics** to prevent repetition
- **Type inference** to reduce redundant annotations
- **Deep typechecking** to prevent errors
- **Resource management** for runtime correctness
- ✖ **Unnecessary overhead**

Language development examples:

- New features of C++
- Rust, Nim, D, Swift, . . .
- Ivory (embedded in Haskell)
- C verified by Isabelle/HOL (SeL4 kernel)

Previously in a bachelor thesis: CHM

- "C with type inference"
  …based on Hindley-Damas-Milner system
- Overloading and generics
  via typeclasses
- Monomorphization
  (similar to C++ template instantiation)

Previously in a bachelor thesis: CHM

- “C with type inference”
  …based on Hindley-Damas-Milner system
- Overloading and generics via typeclasses
- Monomorphization
  (similar to C++ template instantiation)

- ✖ Poor inference of `struct` member access
  (What is the type of `.some_member`?)
- ✖ No support for resource management
  (e.g. locks have to be unlocked, files closed)
- ✖ No subtypes
  (C `const`, implicit number conversions)
- ✖ Not much extensibility
  (e.g. existentials for run-time polymorphism)
- ✖ Code generation limited to C

Previously in a bachelor thesis: CHM

- "C with type inference"
  ...based on Hindley-Damas-Milner system
- Overloading and generics via typeclasses
- Monomorphization
  (similar to C++ template instantiation)

- ✖ Poor inference of `struct` member access
  (What is the type of `.some_member` ?)
- ✖ No support for resource management
  (e.g. locks have to be unlocked, files closed)
- ✖ No subtypes
  (C `const`, implicit number conversions)
- ✖ Not much extensibility
  (e.g. existentials for run-time polymorphism)
- ✖ Code generation limited to C

Idea for this thesis: Try a better type system and see what happens.

CHM record fields

```
/* CHM required explicit type
   variable declarations like C++
*/
<a : Has_x<a>>
void access(a s) {
  // what type is this:
  s.x;
}
```

CHM (previous work) solution:

- All fields of the same name have to share the same type:
  `struct A { int x; };`
  `struct B { ... x; };`
  The type in `B` required to be `int` too
- ✖ Field types are still not inferred
- Field access is `struct` $\rightarrow$ `field`
- We want to overload in both types, and for a given `struct`, we want the **`field`** type be unique
  ➜ multi-parameter typeclasses with functional dependencies (MPTCs)

Main results in this thesis

- ✔ Substantial subset of `C--` and LLVM assembly output
- ✔ Reduction of code repetition via generics
- ✔ Capable constraint-solver-based type system
- Support for multiparameter typeclasses allows:
  - ✔ Modeling `struct` member access
  - ✔ Resource management via `Drop` typeclass, which defines the `drop` function
- ✔ Extensible subtype system (`C--` constness and data 'kinds')

An example of using generics:

```
// User-provided
//   polymorphic definition
add(auto x, auto y)
{ return (x + y); }

bits32 a1, b1; auto c1;
a1 = 5; b1 = 10;
c1 = add(b1, c1);

bits64 a2, b2; auto c2;
a2 = 5; b2 = 10;
c2 = add(b2, c2);
```

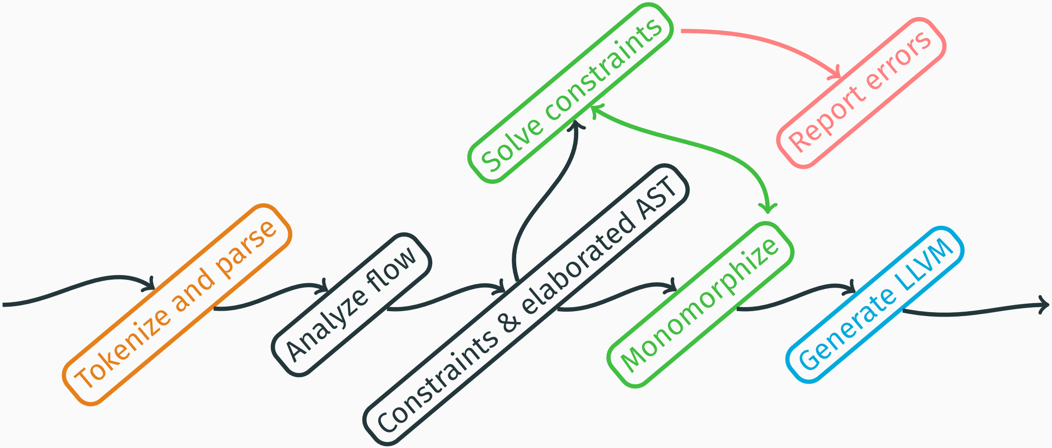
```
// Automatically generated
//   monomorphic definitions
_Madd_$f$Lb32...(bits32 x, bits32 y)
{ return (x + y); }

_Madd_$f$Lb64...(bits64 x, bits64 y)
{ return (x + y); }

bits32 a1, b1; bits32 c1;
a1 = 5; b1 = 10;
c1 = _Madd_$f$Lb32...(a1, b1);

bits64 a2, b2; bits64 c2;
a2 = 5; b2 = 10;
c2 = _Madd_$f$Lb64...(a2, b2);
```

Difference between `C--` and `C`

- Non-extended `C--` has very primitive type system
  - However, with subtypes that affect register allocation and ‘constness’ of variables
- `C--` contains advanced control-flow primitives
  - Explicit tail-calls: `jump` statements, contrasting `call` statements
  - Interprocedural equivalents to `goto` statements: `cut to` statements
- `C--` has a simpler structure, for example:
  - No `while` and `for` loops
  - The whole function is one scope, variable declarations hoisted
  - Expressions have no side-effects
- `C--` allows for various alignment and aliasing assertions

Compiler

Proof-of-concept compiler



Compiler implemented in Haskell, uses `llvm-hs-pure` and `megaparsec`; constraint-based type inference is implemented natively.

Tokenization and parsing

- Use the `megaparsec` library
- Generate the AST extending the `C--` language syntax specification from:
  `https://www.cs.tufts.edu/~nr/c--/extern/man2.pdf`
  In-code documentation clearly indicates all deviations from the standard so the implementation could be used for pure `C--`

Flow analysis – “Flattening” and “Blockifying”

- Splitting into basic-blocks – to facilitate flow analysis and LLVM generation
- Generation of the control-flow graph for each function
- Analysis of variable live-ranges inside the functions
- Insertion of resource management calls to the `drop` function (similar to use of destructors)

Constraint generation and AST elaboration

- Input: the AST of the code
- Output
  - List of constraints to be solved
  - The elaborated AST
- The constraints capture the desired type semantics of the code
- Example (simplified):

$$x = a + b;$$

  - $x : \alpha;\ a : \beta;\ b : \gamma;\ a + b : \delta;$ AST elaborations
  - $\mathtt{Num}\ \delta$ the result belongs to the `Num` typeclass (which defines (+))
  - $\beta \sim \delta; \gamma \sim \delta$ arguments have the type of the result
  - $\alpha \sim \delta$ the assignee and the result share the type

Constraint solving

- Input: list of constraints the types have to satisfy
  For example:
  - Equality constraint $\tau_1 \sim \tau_2$ types τ_1, τ_2 have to be the same
  - Class constraint $C\tau$ the type τ has to belong to the typeclass C
- Output: assignments of type variables to concrete types, leftover constraints
- We can combine the constraints C_1, C_2 (conjunction)

$$C_1, C_2$$

- And nest the constraint C while binding certain type variables a, b and providing local assumptions A

$$\forall a, b. A \Rightarrow C$$

Monomorphization

- Input: AST elaborated with the inferred (poly-)types
- Output: monomorphized AST elaborated with monotypes
  - Monomorphization = replacing polymorphic definitions with copies with the intended monotypes

LLVM generation

- Input: monomorphized AST
  - Elaborated with monotypes
  - Split into basic-block structure
  - Control-flow and variable liveranges in relation to basic-blocks
- Output: the LLVM assembly
- Regular variables translated into LLVM registers
- Each local stack-object `alloca`'ted at function entry
- Translating each basic block separately
  - Variables assigned-to by other basic blocks (or itself) imported via ϕ-nodes
  - Mapping variables to the current values while translating the inner statements

LLVM generation

- Input: monomorphized AST
    - Elaborated with monotypes
    - Split into basic-block structure
    - Control-flow and variable liveranges in relation to basic-blocks
- Output: the LLVM assembly
- Regular variables translated into LLVM registers
- Each local stack-object `alloca`'ted at function entry
- Translating each basic block separately
    - Variables assigned-to by other basic blocks (or itself) imported via ϕ-nodes
        - Located at the basic block entry
        - For each variable that has to be alive
        - Each ϕ-node takes all the possible values of the imported variables
        - The values accompanied with the corresponding exporting basic-block labels
    - Mapping variables to the current values while translating the inner statements

LLVM generation

- Input: monomorphized AST
  - Elaborated with monotypes
  - Split into basic-block structure
  - Control-flow and variable liveranges in relation to basic-blocks
- Output: the LLVM assembly
- Regular variables translated into LLVM registers
- Each local stack-object `alloca`'ted at function entry
- Translating each basic block separately
  - Variables assigned-to by other basic blocks (or itself) imported via ϕ-nodes
  - Mapping variables to the current values while translating the inner statements
    - Each statement can change this mapping
    - This mapping then taken as the basic block's export

Details about the type system

Type system details

- Parametric polymorphism with definable type constructors
- Multi-parameter typeclasses: overloading constrained by a relation
- Automatic calls to resource management functions (bound to a resource lifetime)
- New subtype extension to Hindley-Milner-style type systems

Type system details

- Parametric polymorphism with definable type constructors
  - Types: `bits32`, $\alpha \to \beta$, $\forall a.a$
  - Type variables
    - Free (unbound): $\forall a.a \to \alpha$
      can be later *substituted* with a different type
    - Bound: $\forall a.a \to a$ (`id` function)
      can be instantiated into multiple types according to the given context
  - Type constructors
    - Functions: $\tau \to \sigma$
    - Type constants: `bits32`, `bool`
    - Product types, sum types, and their combination
    - `struct` definitions

Type system details

- Multi-parameter typeclasses: overloading constrained by a relation
  - Multi-parameter typeclass: a relation over some types (represented by type variables)

    `class Name a b c` a typeclass over types a, b, c

    `instance Name bits32 bits32 bits64` a typeclass instance given by $[a := \texttt{bits32}, b := \texttt{bits32}, c := \texttt{bits64}]$
  - Typeclass method: a function defined under the given typeclass, their type defined via the type variables of the enclosing typeclass

    `m :: (a, b) -> c`: a method with the argument (a, b) returning c

    The class instance contains the instance of $\texttt{m}$ typed $(\texttt{bits32}, \texttt{bits32}) \to \texttt{bits64}$
  - Functional dependencies: `class Name a b c | a b -> c` - specifies that, for any instance, each pair a, b uniquely determines c (we can deduce it)

Type system details

- Automatic calls to resource management functions (bound to a resource lifetime)
  - Closely related to the language's semantics, here we assume `C--`: a scope is defined by a function activation (from the function entry to one of the corresponding exits); as customary, we tie a resource liverange to a scope
  - Resource management actions performed by the user-definable `drop` functions
    - usually overloaded in a `Drop` class so that every object issues a different action
  - resource-representing objects defined in `stackdata { ... }` blocks

    `Name: someType;` Name is a pointer to a stack-allocated object of `someType`

    `Name: new someType;` ... and the object represents a resource ➔ `drop`
  - Automatically inserted calls to the `drop` function at function exits (`return`s)
    - Unless such exit follows a `dropped` statement that takes the name of the resource object
    - In C++, for example, scopes with multiple exit points are the minority

Type system details

- New subtype extension to Hindley-Milner-style type systems
  - Many systems programming languages have subtypes (`const`, `constexpr`, etc.)
  - In the extended `C--`: *kinds* and *costnesses*
  - Kind: a set of registers a value can be stored on
  - Constness: specifies when the value is to be computed
    compile-time, link-time, run-time
  - Each HM-style type is extended with a kind dimension and a constness dimension
  - Type (in-)nequalities can be specific to a dimension $\tau_1 \leq_{const} \tau_2$
  - Different types can have a defined inequality relation in each dimension and yet be incomparable
  - Different types can be equal in every dimension and yet still be nonequal

Thank you for attention!

You can try the results on GitHub:

`https://github.com/jiriklepl/masters-thesis-code`

Questions

Backwards-compatibility and extensibility of other languages' type systems

- The extended `C--` is not fully backwards-compatible with `C--` as it uses stronger type system
    - Compiling `C--` programs would be possible with adding some type coercions
    - `C--` programs are not meant to be portable between implementations anyway
- The type system as explained in the thesis does not fit dynamic languages like Python, JavaScript or Julia without significantly restricting such languages
    - Extending the type system with existentials as discussed in the thesis would make modelling such languages easier – however, still not complete
- The type system could be modified to fit the needs of a Rust-like language
    - Successful implementation of Rust borrowing through type system would require the type system to be flow-sensitive