Generating Incremental Type Services

André Pacak
JGU Mainz, Germany

Sebastian Erdweg
JGU Mainz, Germany

Abstract

In this *vision paper*, we propose a method for generating fully functional incremental type services from declarations of type rules. Our general strategy is to translate type rules into Datalog, for which efficient incremental solvers are already available. However, many aspects of type rules don't naturally translate to Datalog and need non-trivial translation. We demonstrate that such translation may be feasible by outlining the translation rules needed for a language with typing contexts (name binding) and bidirectional type rules (local type inference). We envision that even rich type systems of DSLs can be incrementalized by translation to Datalog in the future.

CCS Concepts • **Theory of computation → Type structures**; **Program analysis**.

Keywords incremental, bidirectional type checking

ACM Reference Format:
André Pacak and Sebastian Erdweg. 2019. Generating Incremental Type Services. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE '19), October 20–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3357766.3359534

1 Introduction

Static typing helps developers to debug their code, and it helps IDEs to provide semantic editor services like code completion. To be effective, type information needs to be available immediately while developers edits code. However, DSLs in particular often lack such incremental type services because the development effort for these services is too high.

Our goal is a technique for generating incremental type services from declarations of type rules. An incremental type service must not only react to code changes quickly, but also allow various type-related queries, for example:

1. Given an expression, find its type.
2. Given an expression and a type, check conformance.
3. Given a type, find variables in scope of that type.
4. Given a type, find expressions of that type.

Existing generic solutions to incremental type checking [2, 6, 13] suffer two problems. First, they support a limited set of type features only, e.g. struggling with transitive subtyping. Second, they only support basic typing queries like (1.) and (2.) but not queries of editor services like (3.) and (4.).

We propose to generate incremental type services by compiling type rules to Datalog. Datalog [4] is a logic programming language that can be used to derive relations from input facts via logical rules. Efficient incremental solvers already exist for Datalog, updating the relations when the input facts change [10, 11]. In our setup, the input facts describe the user code while the derived relations describe the code's name and typing information. We believe our approach can avoid the problems from above. First, Datalog is a flexible language in which many computations can be embedded easily, for example encoding subtyping as the transitive closure over inheritance. Second, Datalog relations can be queried in any direction, allowing us to answer all queries introduced above.

But Datalog and its incremental solvers also come with specific requirements, which prevent a natural translation of type rules to Datalog. Specifically, Datalog relations must range over finite domains. However, the typing relation ranges over typing contexts `Ctx` and types `Type`, both of which are infinite domains in most languages. The domain of `Type` is infinite due to structural types like pair, record, struct, and function types. The domain of `Ctx` is infinite because it contains `Type`. Only the names in `Ctx` and the expressions `Exp` are finite because they are restricted to fragments of the current user code. If we want to compile type rules to Datalog, we need to find a way to avoid the infinite domains `Ctx` and `Type`.

The contributions of this vision paper are:

- We propose the generation of incremental type services by compilation of type rules to Datalog.
- We identify the challenges of this approach, namely we need to avoid the infinite domains `Ctx` and `Type`.
- We propose translation steps for type rules that preserve their meaning yet make them compatible to Datalog. We illustrate the translations for a bidirectional type checker of the simply typed lambda calculus.

2 A Bidirectional Type Checker

We introduce a simple type system to illustrate the challenges that occur in the translation to Datalog and to showcase how we solve these challenges. Specifically, Listing 1 shows a bidirectional type checker for the simply typed lambda calculus (STLC) with numbers implemented in Haskell. A bidirectional type checker [7] utilizes two mutually recursive

SLE '19, October 20–22, 2019, Athens, Greece

ACM ISBN 978-1-4503-6981-7/19/10…$15.00
https://doi.org/10.1145/3357766.3359534

```
data Term = Var Name | App Term Term |
  Lam Name Term | Anno Term Type |
  Zero | Succ Term
data Type = Fun Type Type | Nat
data Ctx = Empty | Bind Ctx Name Type

data Typed a = Typed a | Error String
instance Monad Typed where ...

lookup :: Ctx → Name → Typed Type
lookup Empty x = fail "Unbound␣variable"
lookup (Bind c x t) y | x == y = return t
                      | otherwise = lookup c y

infer :: Ctx → Term → Typed Type
infer ctx Zero = return Nat
infer ctx (Succ t) = do
  check ctx t Nat
  return Nat
infer ctx (Var x) = lookup ctx x
infer ctx (Anno t ty) = do
  check ctx t ty
  return ty
infer ctx (App t1 t2) = do
  ty ← infer ctx t1
  (ty1, ty2) ← matchFun ty
  check ctx t2 ty1
  return ty2
infer _ t = fail "Cannot␣infer"

check :: Ctx → Term → Type → Typed ()
check ctx (Lam x t) ty = do
  (ty1, ty2) ← matchFun ty
  check (Bind ctx x ty1) t ty2
check ctx t ty = do
  ty' ← infer ctx t
  matchType ty ty'

matchFun :: Type → Typed (Type, Type)
matchFun (Fun ty1 ty2) = return (ty1, ty2)
matchFun _ = fail "Type␣is␣not␣a␣function"

matchType :: Type → Type → Typed ()
matchType expect given | expect==given = return ()
matchType _ _ = fail "Types␣do␣not␣match"
```

Listing 1. Bidirectional type checker of STLC with numbers.

typing judgments, a `check` and an `infer` judgment. While `infer` tries to compute the type of a term given a typing context, `check` validates a term against a given type. We chose a bidirectional type checker to show that our approach can be applied to non-trivial type systems including type inference.

We describe the bidirectional type checker using a monad `Typed` that either yields a value or an error. Functions `lookup` and `infer` yield a `Type` or an error, function check yields the unit value `()` or an error. With this monad, each statement in `infer` and `check` corresponds to a premise of the corresponding type rules. For example, we can infer the type of applications `(App t1 t2)` by (i) inferring the type `ty` of `t1`, (ii) matching `ty` to ensure it is a function type `(Fun ty1 ty2)`, (iii) checking that `t2` has type `ty1`, and (iv) yielding the result type `ty2`. If one of these steps does not succeed, `infer` will yield the first error it encounters.

Based on this type checker, we can better explain the challenges involved in translating to Datalog. Datalog programs describe relations and incremental Datalog solvers eagerly enumerate all tuples of these relations. But such enumeration can only terminate if the relation inputs come from finite domains. However, `Type` and `Ctx` define infinite domains and therefore cannot be used as relation inputs. This precludes a direct translation to Datalog for our functions `lookup`, `infer`, or `check`. Note that while `Term` also defines an infinite domain, in practice we will only consider a particular instance of `Term`, namely the current user program. Infinite domains in function outputs are not problematic because of their functional dependency [8] on the inputs.

In the next section, we will show how to transform the type checker into one that avoids infinite domains as function inputs. But as first step we will instrument the type checker to find *all* type errors in a program, not just the first one. This is useful for providing feedback anyway, but it also yields better incremental performance because if the user fixes one type error, the remaining `Typed` results remain valid.

3 Translation to Datalog

In this section, we will show how we tackle the previously identified challenges by transforming the type checker.

3.1 Computing All Types and Type Errors

The type checker from Listing 1 stops as soon as it finds a type error. This can be seen in the single error message that `Error` accepts. However, to enable a wide range of type services, we want to compute all types and type errors of a program and continue even after the first type error was found. This is what our first translation step does.

We introduce an auxiliary type `Any` that we use whenever the actual type cannot be computed. The use of `Any` is inspired by gradual typing, where the static type checker also uses such a catch-all type for the parts of the code that don't have a static type [3]. In our type checker, we want to use `Any` whenever type inference fails. For example, in the case for function application `App`, when type inference of `t1` fails, we want to approximate its type as `Any` so that we can continue checking `t2`. We can introduce this behavior by adopting the monad for `Typed` to continue after an error using a catch-all value `top` (which is `Any` for `Type`):

```
data Type = Fun Type Type | Nat | Any
data Typed a = Typed a | Error [String]
instance RMonad Typed where
  type RMonadCtx Typed a = WithTop a
  return a = Typed a
  t>>=f = case t of
    Typed ty → f ty
    Error err1 → case f top of
      Typed _ → Error err1
      Error err2 → Error (err1 ++ err2)
```

We use a restricted monad RMonad that only accepts types that are instances of type class WithTop. In the bind function >>=, when t is an error, we call f anyway using the top value. We make sure to propagate any errors encountered afterwards.

Down the line, we are using matchFun to deconstruct function types and matchType to compare types. We have to adapt these functions to take the new Any type into account. As in gradual typing, the Any type can be treated as a function (Fun Any Any). When comparing types, we use the ordering ≥ to ensure the given type is at least as specific as the expect type. Type Any is larger than all other types and, in contrast to subtyping, ≥ is covariant for functions in the argument and result type.

```
matchFun :: Type → Typed (Type, Type)
matchFun (Fun ty1 ty2) = return (ty1, ty2)
matchFun Any = return (Any, Any)
matchFun _ = fail "Type is not a function"

matchType :: Type → Type → Typed ()
matchType expect given | expect≥given = return ()
matchType _ _ = fail "Types do not match"
```

For the App case, when infer ctx t1 fails with errors, we continue with the Any type for ty. The call to matchFun ty will then yield Any for both ty1 and ty2. We thus check the function argument against Any, which will succeed as long as the argument is typeable at all. If the check yields further type errors, we propagate those together with the errors of the infer call.

3.2 Eliminating the Type Argument

Our type checker has two kinds of infinite input domains: types and contexts. In this translation step, we eliminate the type input; we deal with contexts in the next subsection.

Function check is the only function that receives a type as input. The type prescribes the expected type of an expression. Our idea is to reverse the data flow of type arguments: Instead of passing a type down, we require the type from above when we need it. Specifically, we introduce a new function

```
required :: Ctx → Term → Typed Type
```

that computes the type that is required of a checked expression. We rewrite check_{old} to check_{new} by calling required whenever the input type is needed, such that:

```
check_new c e = check_old c e (required c e)
```

The difficulty lies in generating required. The idea is that required reconstructs the type argument of check_{old}, but on demand. To this end, required contains one case for each *call site* of check_{old}. We show the derived code of required in Listing 2 and explain each case below.

The first call of check_{old} in Listing 1 is in the Succ case of infer. The code says that sub-expression t of (Succ t) must have type Nat. This call site becomes a case in required stating that the required type of t is Nat if the parent of t is

```
required :: Ctx → Term → Typed Type
required ctx term = case parent term of
  Just (Succ t _) | term == t → return Nat
  Just (Anno t ty _) | term == t → return ty
  Just (App t1 t2 _) | term == t2 → do
    ty ← infer ctx t1
    (ty1, _) ← matchFun ty
    return ty1
  Just (Lam x t p) | term == t → do
    let (Bind ctx' _ _) = ctx
    ty ← required ctx' (Lam x t p)
    (_, ty2) ← matchFun ty
    return ty2
```

Listing 2. Computing the required type for check.

(Succ t). The call site of check_{old} for Anno can be translated similarly, but yielding the annotated type.

The call site of check_{old} for (App t1 t2) is more complicated because the type argument is computed. Specifically, we call infer and matchFun to determine the argument type of t1, against which we check the argument expression t2. This call site illustrates the general translation scheme we use to derive the cases of required:

1. Match the parent term to identify the call site of check belonging to the current term.
2. Compute the program slice for the type argument of the call site.
3. Copy the program slice into required.
4. Return the type argument as result of required.

For App, this means we copy the calls of infer and matchFun into required, yielding the argument type as result.

The case for Lam follows the same scheme, but has to deal with two specific issues. First, the invocation of check for Lam was a recursive call. Therefore, when computing the program slice, we need to replace any reference to check's type argument with a recursive call of required on the parent. The second issue is that the call of check was passed an extended context. However, we need to interpret the slice using the original context. To this end, we invert the context construction by deconstructing it again, yielding ctx'.

Based on required, we obtain check_{new} code that behaves the same as check_{old} yet does not need a type argument:

```
check :: Ctx → Term → Typed ()
check ctx (Lam x t p) = do
  ty ← required ctx (Lam x t p)
  (ty1, ty2) ← matchFun ty
  check (Bind ctx name ty1) t
check ctx t = do
  ty ← required ctx t
  ty' ← infer ctx t
  matchType ty ty'
```

3.3 Eliminating the Context Argument

To run the type checker in Datalog, we finally need to eliminate the context argument. We eliminate the context by

applying a technique that has been known for reference attribute grammars for a while. The key idea is to define `lookup` as a function that searches for a variable's binding by walking up the tree [5][9, Chapter V]. Such a lookup function does not require a typing context as input, which can thus be eliminated. While the idea is not novel, we are the first to explore how to generate a bottom-up lookup function from a traditional type checker.

Traditionally, functions `check` and `infer` traverse the tree top-down while building up a typing context. That typing context is eventually used to look up the type of variable references. The bottom-up lookup function does the inverse: It starts at the variable reference, walks up the tree, and resolves a variable's type when reaching the binding. We show the derived bottom-up lookup function in Listing 3 and explain how it was derived below.

The derived lookup function takes the term, in which we search for a binding bottom-up, and the name of the variable as input. We derive the bottom-up lookup function based on the call sites of `check` and `infer`. The call sites for the subterms of `Succ`, `Anno`, and `App` pass the context unchanged. Inversely, the bottom lookup function continues searching for a binding in the parent term.

The call site for the body of a `Lam` is more interesting because we use an extended context. This allows us to illustrate the general translation scheme:

1. Match the parent term to identify the call site of `check` or `infer`.
2. Compute the program slice for the context argument of the call site.
3. Copy the program slice into `lookup`.
4. If the context was extended, check if the variable reference is bound here. If it is, return the type that was put in the context for it.
5. If the variable's binding was not found yet, continue in term's parent.

If the current term has no parent, we reached the root of the program and lookup fails.

4 Discussion and Open Challenges

We have proposed three transformations steps for translating a type checker into a Datalog-compatible one. The final type checker is compatible with Datalog because the functions only take finite domains as input for any given program. To confirm this compatibility, we reimplemented the final type checker in a Datalog dialect called IncA [10, 11]. While we don't have space to show the IncA code, the translation was eased by IncA's notation, which resembles functional programming. The only difference to the type checker shown here is that type errors (invocations of `fail`) are collected in a separate AST traversal. We have manually tested the IncA type checker and confirmed that type results are updated incrementally when the input program changes. Thus, we have successfully derived incremental type services for the bidirectional simply typed lambda calculus.

```
lookup :: Term → Name → Typed Type
lookup t x = case parent t of
  Just (Succ t p) → lookup (Succ t p) x
  Just (Anno t ty p) → lookup  (Anno t ty p) x
  Just (App t1 t2 p) | t == t1 →
    lookup  (App t1 t2 p) x
  Just (App t1 t2 p) | t == t2 →
    lookup  (App t1 t2 p) x
  Just (Lam x' t p) →
    if x == x'
      then do
        ty ← required (Lam x' t p)
        (ty1, _) ← matchFun ty
        return ty1
      else lookup (Lam x' t p) x
  Nothing → fail ["Unbound variable"]
```

Listing 3. The derived bottom-up `lookup` function.

We envision that our approach can work to derive efficient, incremental type services in general. To this end, we identify the following challenges:

Generalization. We have shown how to derive incremental type services for a single language so far. We want to study to which languages we can generalize the proposed method. For example, we want to investigate how to support subtyping, records, objects, nominal typing, type classes, dependent types, and other features.

Type system DSL. We would like to define a domain-specific language that can be used to declare rich type systems. This DSL would be the basis for a compiler that has IncA as a compilation target. We will draw inspiration from existing type system DSLs [1, 12].

Transformation mechanization. The transformations outlined in the previous section have not yet been rigorously spelled out. To clarify the exact prerequisites and behavior of the transformations, we will mechanize their definition. We hope this will also enable us to validate that the transformations preserve the type system semantics.

Performance. Finally, we want to evaluate the performance of our derived Datalog type services. We are mostly interested in the incremental performance: How long does it take to receive fully updated typing information after a program change.

Acknowledgments

We thank the anonymous reviewers for their feedback on this paper. Additionally, we would like to thank our colleagues Sven Keidel and Tamás Szabó for their useful feedback.

References

[1] Lorenzo Bettini, Dietmar Stoll, Markus Völter, and Serano Colameo. 2012. Approaches and Tools for Implementing Type Systems in Xtext. In *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*

(Lecture Notes in Computer Science), Krzysztof Czarnecki and Görel Hedin (Eds.), Vol. 7745. Springer, 392–412. https://doi.org/10.1007/978-3-642-36089-3_22

[2] Sebastian Erdweg, Oliver Bracevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. 2015. A co-contextual formulation of type rules and its application to incremental type checking. In *OOPSLA*. ACM, 880–897.

[3] Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting gradual typing. In *POPL*. ACM, 429–442.

[4] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. 2013. Datalog and Recursive Query Processing. *Foundations and Trends in Databases* 5, 2 (2013), 105–195.

[5] Görel Hedin. 2000. Reference Attributed Grammars. *Informatica (Slovenia)* 24, 3 (2000).

[6] Edlira Kuci, Sebastian Erdweg, Oliver Bracevac, Andi Bejleri, and Mira Mezini. 2017. A Co-contextual Type Checker for Featherweight Java. In *ECOOP (LIPIcs)*, Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 18:1–18:26.

[7] Benjamin C. Pierce and David N. Turner. 1998. Local Type Inference. In *POPL*. ACM, 252–265.

[8] Kenneth A. Ross and Yehoshua Sagiv. 1992. Monotonic Aggregation in Deductive Databases. In *PODS*. ACM Press, 114–126.

[9] Emma Söderberg. 2012. *Contributions to the Construction of Extensible Semantic Editors.* Ph.D. Dissertation. Lund University.

[10] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing lattice-based program analyses in Datalog. *PACMPL* 2, OOPSLA (2018), 139:1–139:29.

[11] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: a DSL for the definition of incremental program analyses. In *ASE*. ACM, 320–331.

[12] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. 2018. Scopes as types. *PACMPL* 2, OOPSLA (2018), 114:1–114:30. https://doi.org/10.1145/3276484

[13] Guido Wachsmuth, Gabriël D. P. Konat, Vlad A. Vergu, Danny M. Groenewegen, and Eelco Visser. 2013. A Language Independent Task Engine for Incremental Name and Type Analysis. In *SLE (Lecture Notes in Computer Science)*, Vol. 8225. Springer, 260–280.