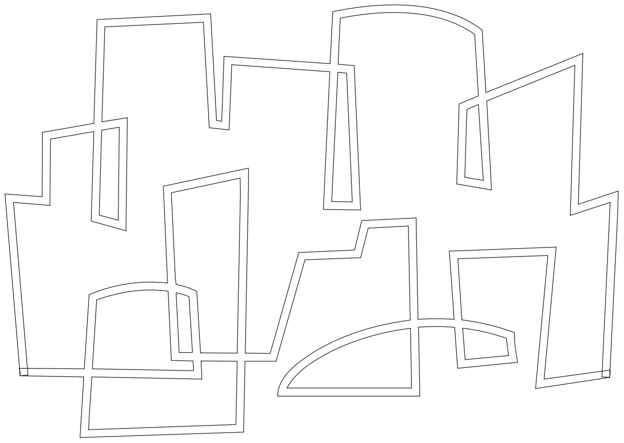
Incremental Polymorphism

Shail Aditya, R.S. Nikhil

In Proceedings on Functional Programming and Computer Architecture, Cambridge, MA, August 28-30, 1991

1991, August

Computation Structures Group
Memo 329



Incremental Polymorphism

Computation Structures Group Memo 329
June 6, 1991

Shail Aditya

Rishiyur S. Nikhil

To appear in Proc. Functional Programming Languages and Computer Architecture, Cambridge, MA, Aug 28-30, 1991

This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-1988.

Incremental Polymorphism

Shail Aditya Rishiyur S. Nikhil

Abstract

The Hindley/Milner polymorphic type system has been adopted in many programming languages because it provides the convenience of programming languages like Lisp along with the correctness guarantees that come with static type-checking. However, programming environments for such languages are still not as flexible as those for Lisp. In particular, the style of incremental, top-down program development possible in Lisp is precluded because the type inference system is usually formulated as a "batch system" that must examine definitions before their uses. This may require large parts of the program to be recompiled when a small editing change is performed.

In this paper, we attempt to strike a balance between the apparently conflicting goals of incremental, top-down programming flexibility and static type-checking. We present an incremental typing mechanism in which top-level phrases can be compiled one by one, in any order, and repeatedly (due to editing). We show that the incremental type system is sound and complete with respect to the more traditional "batch system". The system derives flexibility from the inherent polymorphism of the Hindley/Milner type system and minimizes the overhead of book-keeping and recompilation. Our system is implemented and has been in use by dozens of users for more than two years.

1 Introduction

Modern computing environments strive for several desirable features: the environments should support the development of reliable programs, *i.e.*, they should be able to detect as many programming errors as early as possible; the environments should be robust, *i.e.*, they must gracefully report all errors and exceptions as and when they occur; finally, the environments should have flexible and interactive facilities for editing, testing and debugging of programs.

Strongly typed languages meet the first goal by guaranteeing that "type-consistent" programs will not incur run-time type-errors. Recent programming languages based on the Hindley/Milner type system [4, 8] also provide the convenience of type polymorphism and automatic type inference. But most programming environments for such languages have to compromise the flexibility of incremental,

Current address for both authors: Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Massachusetts 02139. Current Internet e-mail address: `shail@abp.lcs.mit.edu` and `nikhil@abp.lcs.mit.edu`.
This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-1988.

top-down program development in order to achieve this "type-consistency" over the whole program. This is because the Hindley/Milner type system is usually formulated as a "batch system" that must examine the definitions before they can be used and the complete program before any of its parts can be exercised. The problem is further complicated due to polymorphism where the meaning of "type-consistency" is one of inclusion, rather than equality.

In this paper, we will address the issue of providing a robust and interactive programming environment for Id, which is a polymorphic, strongly typed, incrementally compiled, parallel programming language developed at the Laboratory for Computer Science, MIT [11]. Id's typing mechanism is based on the Hindley/Milner type inference system, but it differs from its traditional description in that it is "incremental" in nature. We have modified and extended the standard Hindley/Milner type inference algorithm to facilitate incremental development and testing of programs. In this paper, we will describe this incremental algorithm used in Id. We will also show its correctness (soundness and completeness) with respect to the standard formulation of the Hindley/Milner type inference algorithm that examines the complete program. Our goal is to produce the same typings as obtained *via* the standard algorithm, only that we produce them incrementally. For convenience, we will refer to the traditional type inference mechanism as the "batch system" and our mechanism as the "incremental system" throughout the paper.

The paper is organized as follows. Section 2 discusses a few other interactive programming environments and motivates the incremental approach taken in Id. Section 3 establishes the notations used in this paper. Section 4 describes the issues in incremental type inference under an interactive environment *via* several examples. Section 5 describes our incremental type inference system in detail. In Section 6 we show its correctness. Section 7 extends the incremental algorithm to handle complex editing situations. In Section 8 we discuss the complexity of our system and briefly describe some possible optimizations that are detailed in appendix A. Finally, in Section 9 we summarize our results and compare them with the related work in this field.

2 Background

Nikhil in [10] pointed out the disparity between the goals of an incremental programming environment, and ML-like type inference system. ML [2, 9] is interactive, but a session in ML is essentially a large lexically nested ML program. Each toplevel definition has the rest of the session as its scope. Thus, editing an earlier definition may force the user to recompile and reload all the intermediate definitions that used it[1]. In building large systems, the recompilation and reloading of large pieces of potentially unrelated code, just to recreate the same environment every time a small error is detected, is at best, quite annoying.

In Miranda [16, 17], this problem is resolved by making the unit of compilation to be a whole file (also called a "Miranda script"). Definitions within a file may appear in any order and the compiler is responsible for reordering them during

[1]SML's module mechanism gives some relief in this respect, due to separate compilation.

compilation. The interactive session only evaluates expressions using definitions from the current script. But, definition level incrementality has been lost, and editing forces entire files to be recompiled.

In contrast, Lisp programming environments smoothly integrate the editor, the compiler, and the read-evaluate-print loop. The unit of compilation in these systems is a single top-level definition. They allow the user to furnish multiple top-level definitions, either together, or one by one in any order, resolving global references to other definitions automatically by dynamic linking. The user can test, debug, and edit these definitions incrementally, without waiting to write the complete program or having to recompile a substantial fraction of the definitions already supplied.

In Id, we attempt to achieve the flexibility of Lisp-like environments along with static type-checking. The programming environment for Id, called "Id World"[12], smoothly integrates the editor, the Id compiler, and the underlying execution vehicle. Like Lisp, our unit of compilation is a single top-level definition. Each compiled definition is accumulated into a flat global environment, implying that edited definitions are immediately and automatically made available to other definitions that use them. Simple interprocedural book-keeping maintains type-consistency between the definition and the uses of each top-level identifier. The book-keeping mechanism derives flexibility from the polymorphism of the definitions, flagging only inconsistent definitions for recompilation. As a consequence, we permit out of order compilation, redefinition, and editing with minimum overhead, while still guaranteeing type correctness before program execution. We also allow executing partially defined Id programs, where the undefined identifiers simply generate an exception if actually used at run-time. Again, the incremental book-keeping mechanism helps in incorporating the missing parts as and when they become available with minimum overhead.

3 Syntax and Notation

In this section, we present a brief overview of the notation used in this paper. Readers are referred to [5, 14] for details. Those already familiar with the Hindley/Milner type system may just skim this section for the notation — there are no new concepts introduced here.

3.1 The Expression Mini-Language

The basic expression language used in describing the Hindley/Milner type system is fairly small. We use the same formulation as in [4, 14] with only slight modifications to suit our incremental system. The syntax of the mini-language appears below:

$$x, y, z \in \text{Identifiers} \tag{1}$$

$$c \in \text{Constants} \quad ::= \quad \{\texttt{true}, \texttt{false}, 1, 2, \ldots\} \tag{2}$$

$$
\begin{array}{rcll}
e \in \text{Expressions} & ::= & \quad c & (3)\\
 & & \mid\ x & \\
 & & \mid\ \lambda x.\, e_1 & \\
 & & \mid\ e_1\, e_2 & \\
 & & \mid\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 & \\
B \in \text{Bindings} & ::= & x = e & (4)\\
P \in \text{Programs} & ::= & B_1;\ B_2; \cdots;\ B_n;\ \texttt{it} = e & (5)
\end{array}
$$

The above mini-language retains the notion of top-level, independent bindings as units of compilation. A complete program is a set of such bindings. The final binding is special[2] in that it represents a program query and is also evaluated after being compiled within the current environment.

We will freely (and informally) use tuple expressions and tuple bindings in this mini-language because tuples can always be simulated by the function type constructor ($\rightarrow$) alone.

3.2 The Type Language

The standard Hindley/Milner type language is inductively defined as follows:

$$
\begin{array}{rcll}
\pi \in \text{Type-Constructors} & = & \{\mathit{int}, \mathit{bool}, \ldots\} & (6)\\
\alpha,\beta \in \text{Type-Variables} & = & \{\texttt{*0}, \texttt{*1}, \ldots\} & (7)\\
\tau \in \text{Types} & ::= & \quad \pi & (8)\\
 & & \mid\ \alpha & \\
 & & \mid\ \tau_1 \rightarrow \tau_2 & \\
\sigma \in \text{Type-Schemes} & ::= & \quad \tau & (9)\\
 & & \mid\ \forall\alpha.\, \sigma_1 &
\end{array}
$$

3.3 Type Notation

A **Type Environment** maps identifiers to type-schemes. All type variables of a type τ are considered to be **free** in that type. The quantified type-variables of a type-scheme are taken to be **bound** in that type-scheme and the other type-variables are taken to be **free**. This definition extends pointwise to type environments. We will denote the free type-variables of T by $\mathit{tyvars}(T)$, where T could either be a type, a type-scheme, or a type environment.

A **Substitution** S maps type-variables to types. By extension, we can apply substitutions to types, type-schemes, or type environments, in each case only operating on their free type-variables. Given a type-scheme $\sigma = \forall\alpha_1 \cdots \alpha_n.\, \tau$, an **Instantiation** $\sigma \succ \tau'$ is defined by a substitution S for the bound variables of σ so that $S\tau = \tau'$. The instantiation $\sigma_1 \succeq \sigma_2$ is valid if $\sigma_1 \succ \tau_2$ (where $\sigma_2 = \forall\beta_1 \cdots \beta_m.\, \tau_2$) and no β_j is free in σ_1. The key point to remember is that substitutions affect only free type-variables, while instantiations operate only on bound type-variables.

[2]This is adapted from the ML interactive runtime environment where the last expression evaluated is referred to as "`it`".

Given a type τ and a type environment TE, we define $close(TE, \tau) = \forall \alpha_1 \cdots \alpha_n . \tau$ where $\{\alpha_1, \ldots, \alpha_n\} = tyvars(\tau) - tyvars(TE)$. When $tyvars(TE) = \phi$, we may also write simply $close(\tau)$ instead of $close(TE, \tau)$.

4 Issues in Incremental Typing

In this section, we will describe the various issues that need to be addressed during incremental type inference by means of simple examples[3].

4.1 Forward References

In Id, all top-level definitions are compiled into a single global environment. So, it is possible and convenient to use functions that are defined later, as the following example shows[4]:

```
def f x = (x+1):(g x);                  % int -> (list int)

def g x = x:nil;                        % *0 -> (list *0)
```

We have shown the inferred type for each of the definitions as a comment appearing to the right of the definition[5]. Assuming that the above definitions are compiled in the order of their appearance, the function `f` uses `g` inside its body as a function with type `(int -> (list int))` without actually knowing anything about it.

When `g` is compiled, `f` "sees" its definition and the system should make sure that all its previous uses are "consistent" with its definition. In later sections, we will describe in detail how this consistency check is formulated and verified. For now, it suffices to say that use of an identifier should have a type that is an instance of the type inferred at its definition. In the above example, it is indeed the case, the type of `g`, `(int -> (list int))`, used within `f` is an instance of the defined type of `g`, `(*0 -> (list *0))`[6]. Therefore, `f` need not be recompiled even though it used `g` before it was defined.

4.2 Editing

Going a step further, we may edit `g` as follows:

```
def g x = x:(x-1):nil;                  % int -> (list int)
```

This restricts the type of `g`, but its use inside `f` is still valid, and no recompilation is necessary. On the other hand, if we had redefined `g` as,

[3]Even though all our analysis is based on the mini-language given in section 3, our examples use the full Id syntax for convenience and clarity. In [5], we show a simple translation from Id to this mini-language.

[4]In Id, toplevel function definitions are introduced with the keyword `def` and terminated by a semi-colon (`;`). Also, colon (`:`) is the infix cons operator. Text following a percent (`%`) is taken to be a comment and is ignored.

[5]The type variables appearing in a type are assumed to be implicitly universally quantified at the outermost unless otherwise stated or clear by the context.

[6]This instantiation uses the simple substitution $S = \{\texttt{*0} \mapsto \texttt{int}\}$ for the bound type-variable `*0`.

```
   def g x = x-1;                             % int -> int
```

then the inferred type of g no longer matches its use inside f and the system should detect it and report an error. Note that this analysis is independent of the order of the original definition of f and g, or for that matter, any other definitions that appear temporally in between the definitions of f and g. Such independent definitions are never disturbed; all recompilation requirements, if any, apply only to the definitions that fail the consistency check.

4.3 Mutual Recursion

The situation becomes more complicated with mutually recursive functions, which have to be type-checked together in the Hindley/Milner type system. In the incremental system, such definitions may be compiled separately. We have to either rule out such cases by requiring that mutually recursive definitions be supplied together, or incrementally detect definitions that become mutually recursive and handle them appropriately. Simple book-keeping, as described in section 4.1, may fail to catch type-errors embedded across mutually recursive definitions, as the following example shows:

```
   def f x = g f;                             % *0 -> *1
   ...
   def g x = f g;                             % *2 -> *3
```

The above two definitions are mutually recursive and are rejected by the batch system[7]. The incremental system infers the type of f to be (*0 -> *1), as shown above, assuming the type of g to be (*0 -> *1) -> *1). Similarly, when g is compiled, its type is obtained as (*2 -> *3), assuming the type of f to be ((*2 -> *3) -> *3). Note that in both cases, the assumed types are instances of their corresponding inferred types, and the simple consistency checking used in section 4.1 is not sufficient to catch the type-error in the above program.

The following example shows that without explicit book-keeping of mutually recursive definitions, the incremental system may, in fact, compute unsound types even when there is no overall type-error.

```
   def K x y = x;                             % *0 -> *1 -> *0

   def f x = (g f) x;                         % *2 -> *3

   def g x = K x (x:(f x));                   % *4 -> *4
```

The K function simply returns its first argument. The inferred types of f and g individually are as shown. The uses of both f and g are instances of their inferred types, so no consistency error is present. But when supplied together and taking into account that f and g are mutually recursive, the correctly computed Hindley/Milner type of g should be ((*4 -> (list *4)) -> (*4 -> (list *4))) and that of f should be (*4 -> (list *4)).

[7]The Hindley/Milner type system does not handle infinite types and flags them as type-errors. In this example, the type of both f and g are infinite.

4.4 Editing and Type Relaxation

It is possible during editing that the type of a definition gets relaxed and therefore must be reflected in other definitions that use it. Consider the following example:

```
def f x y = (x+1);                  % int -> *0 -> int

def g x y = f x (y+1);              % int -> int -> int
```

The type of `f` constrains the type of `g`'s argument `x` to be `int`. Now, if we decide to edit the function `f` so as to relax its type,

```
def f x y = x;                      % *1 -> *0 -> *1
```

the type of `g` that was earlier constrained to be `(int -> int -> int)` can now also be relaxed to `(*2 -> int -> *2)`. The system should be able to detect this as well. Note that this relaxation does not create unsound types but may render the original types as incomplete or non-principal.

Constraint relaxation may also occur due to a change in mutual recursion among definitions. Consider the following example:

```
def fst (x,y) = x;                  % (*0,*1) -> *0

def h x = if true then
            x
          else fst (t x x);         % *2 -> *2

def t x y = h x,h y;                % *2 -> *2 -> (*2,*2)
```

Since `h` and `t` are mutually recursive, the type of the two arguments of `t` are constrained to be the same. Now, if we edit the definition for `h` to be the simple identity function,

```
def h x = x;                        % *2 -> *2
```

then the constraint on the arguments of `t` is no longer present since `h` can now be instantiated differently. Therefore, the type of `t` can be relaxed to `(*3 -> *4 -> (*3,*4))`. The system should be able to detect this and flag the recompilation of `t`.

5 The Incremental Type Inference System

Our incremental system is based on the standard Hindley/Milner inference rules given in the literature [4, 3, 14], so we will not describe those inference rules again. We follow the description of [14] which expresses the rules for Instantiation and Generalization implicitly. This has the advantage of making the inference rules completely deterministic[8], and we always infer a type for an expression instead of a type-scheme[9].

In this section, we will describe our incremental book-keeping strategy and the incremental type inference algorithm that uses it.

[8]This means that exactly one rule will apply to a given expression.

[9]The equivalence of the rules appearing in [4] and those in [14] has been shown in [3].

5.1 Incremental Book-Keeping

The basic idea in incremental compilation is to be able to compute some desired compile-time properties for an aggregate of identifiers in an incremental fashion. This aggregate forms the **identifier namespace** that we operate in. For our purposes, this is the set of all top-level identifiers. Our first step is to define a "property".

Definition 1 *A* **property** $\mathcal{P} = (\mathcal{D}, \sqsubseteq)$ *is characterized by a domain of values* $\mathcal{D}$ *partially ordered*[10] *by the relation* $\sqsubseteq$*. Given two values,* $v_1, v_2 \in \mathcal{D}$ *for a property* $\mathcal{P} = (\mathcal{D}, \sqsubseteq)$*, we say that* v_1 *is consistent with* v_2 *if and only if* $v_1 \sqsubseteq v_2$*.*

The domain of values is simply a syntactic set of values with some structural relationship defined among its elements. The domain must also contains a special element "$\perp$" (read "bottom") that corresponds to the default property value assigned to as yet undefined identifiers in the namespace.

The interdependences among the properties of identifiers at various times during incremental compilation is maintained *via* sets of "assumptions" defined below.

Definition 2 *An* **assumption** $(x, y, v_y) \in \mathcal{A}$ *is a triple consisting of an* **assumer** x*, an* **assumee** y*, and an* **assumed-value** $v_y \in \mathcal{D}$*, for the assumee's property* $\mathcal{P} = (\mathcal{D}, \sqsubseteq)$*.* $\mathcal{A}$ *is termed as the* **assumption domain***. An* **assumption set** A_x *for the assumer* x *is a set of all such assumptions made by* x *and can be written as a map from the assumees to their assumed-values.*

Each assumption domain has an associated consistency checking function. An **assumption check** C *is a predicate that verifies the assumed-value* v_y *of an assumee from a given assumption set against its current value* v'_y *available in the environment for consistency. This check may use the property predicate* $\sqsubseteq$ *for this purpose.*

Assumption domains are characterized by the properties they record and the assumption checks they employ in order to verify consistency. Several assumption domains may be associated with the same property that use different assumption checks. We will see examples of this later on. Also note that an assumption set for an assumer may contain several assumptions for the same assumee corresponding to its various occurrences in the definition of the assumer.

The union of all the property mappings of namespace identifiers to their property values constitutes a **compilation environment**. The sets of assumptions associated with each assumer identifier make up the book-keeping overhead of the compilation environment.

5.2 Overall Plan for Incremental Analysis

The overall scheme for incremental property computation and maintenance appears in Figure 1. Essentially, we process each top-level binding individually, accumulating its assumptions and property values in the current environment.

[10] A partial order on a domain is a reflexive, transitive, and anti-symmetric binary relation on the elements of the domain.