Type-Indexed Rows

Mark Shields *
Oregon Graduate Institute
mbs@cse.ogi.edu

Erik Meijer
Utrecht University
erik@cs.uu.nl

Abstract

Record calculi use labels to distinguish between the elements of products and sums. This paper presents a novel variation, *type-indexed rows*, in which labels are discarded and elements are indexed by their type alone. The calculus, λ^{TIR}, can express tuples, recursive datatypes, monomorphic records, polymorphic extensible records, and closed-world style type-based overloading. Our motivating application of λ^{TIR}, however, is to encode the "choice" types of XML, and the "unordered tuple" types of SGML. Indeed, λ^{TIR} is the kernel of the language XMλ, a lazy functional language with direct support for XML types ("DTDs") and terms ("documents").

The system is built from rows, equality constraints, insertion constraints and constrained, or qualified, parametric polymorphism. The test for constraint satisfaction is complete, and for constraint entailment is only mildly incomplete. We present a type checking algorithm, and show how λ^{TIR} may be implemented by a type-directed translation which replaces type-indexing by conventional natural-number indexing. Though not presented in this paper, we have also developed a constraint simplification algorithm and type inference system.

1 Introduction

Record (and less often, variant) calculi appear in many contexts. Some functional languages incorporate them in conjuction with more conventional tuples and recursive sums-of-products datatypes [11]. They have been used as foundations for object-oriented languages [30]: objects can be modelled by records, and subclassing can be built upon record subtyping. Database query languages often model relations as sets of records, and, because database schema are dynamic, require a particularly flexible type system [5].

*The research reported in this paper was supported by the USAF Air Materiel Command, contract #F19628-96-C-0161, NSF Grant CCR-9970980, and Utrecht University.

In this paper we present a system very much like an extensible, polymorphic record calculus, but with an essential twist: *we discard labels.* Instead of labels, elements of products and sums are distinguished by their type alone. That is, a *type-indexed row* (TIR) is a list of types (possibly with a type variable as tail), from which we form *type-indexed products* (TIPs) and *type indexed co-products* (TICs). The rôle of labels is played by *newtypes*, which introduce fresh type names.

Of course, in a monomorphic setting such a system is straightforward. In the presence of polymorphism, however, we must somehow resolve the paradox of rows indexed by types which are partially or fully unknown, *i.e.*, which contain free type variables.

We developed λ^{TIR} to treat the *regular expression types* of XML [27] and SGML as types in a functional language we are developing called XMλ [16]. XML includes "choice" types of the form $(\tau_1 \mid \ldots \mid \tau_n)$, and SGML includes "unordered tuple" types of the form $(\tau_1 \& \ldots \& \tau_n)$. Neither of these types include any syntactic information, such as labels, to guide a type checker in deciding which summand of a sum, or which permutation of a product, a given term belongs to. Instead, a *1-nonambiguity* condition is imposed, which implies membership of a term in a regular-expression type may be decided by a deterministic Glushkov automaton [4]. In λ^{TIR}, we abstract from this formulation by requiring only that each type in a sum or product be distinct. This allows XML terms to be naturally manipulated within a polymorphic functional programming language.

Serendipitously, we also found λ^{TIR} could naturally encode:

- conventional tuples and recursive sums-of-product datatypes;
- many existing record calculi, both monomorphic and polymorphic, extensible and non-extensible;
- types resembling Algol 68's union types; and,
- the closed-world style of type-based overloading (modulo subtyping) popular in object-oriented languages [8].

XMλ has many of the types mentioned above. The XMλ complier simply translates each into λ^{TIR}, resulting in a compact and uniform compiler. Hence λ^{TIR}'s expressiveness is not merely of theoretical interest, but can also be exploited in practice.

Many of the ingredients of λ^{TIR} are well known:

- We use a *kind system* to distinguish rows from types.
- As in record calculi, we require *insertion constraints* to ensure the well-formedness of rows, only now they state

that a *type* may be inserted into a row.
- Unlike in record calculi, we also require *equality constraints*, as sometimes the unification of two rows must be delayed if there is any ambiguity as to the matching of their element types.
- Constrained polymorphism [12, 18] is used to propagate constraint information throughout the program, thus ensuring soundness.
- We eagerly test for the unsatisfiability of constraints so as to reject programs as early as possible.
- As in Gaster and Jones' record calculus [7], λ^{TIR} is implemented by a type-directed translation which replaces type-indexing by natural-number indexing. These indices propagate via implicit parameters at run-time to parallel the propagation of insertion constraints at compile-time.

We first review record calculi (§2), then motivate the introduction of each of the above components by small examples (§3). A larger example is also presented (§4). We then develop a type-checking system for λ^{TIR} which simultaneously performs a type-directed translation into an untyped run-time language (§5). This requires the notion of *constraint entailment* (§5.4). We also demonstrate our system is sound (§5.5).

We have also developed a type-inference system for λ^{TIR} which builds upon a constraint simplifier. All the details may be found in the full paper [25].

2 Review: Label-Indexed Rows

To aid the transition to λ^{TIR}, we first quickly review existing calculi of labelled records and variants. We use a somewhat unorthodox syntax, though none is particularly standard anyway. We assume an ambient type system and a set of *label names*.

Rows

We first introduce *rows* [30], which are lists of labelled types. For example:

```
(xCoord: Int) # (yCoord: Int) # Empty
```

is a row with label names `xCoord` and `yCoord`, both labelling type `Int`. Here we use the `#` operator to denote *row extension*, and `Empty` to denote the empty row. (Note that in this paper we shall assume labels are formed from label names by appending a ':'.)

Sometimes *row concatenation* replaces or augments row extension [9], though we do not consider this here.

Rows are equal up to a permutation of their labelled types. That is, the elements of a row are distinguished by their label name rather than by their position.

A record calculus is *extensible* if a row may end with a type variable instead of just `Empty`. For example:

```
(xCoord: Int) # (yCoord: Int) # a
```

is an *open row*, with *tail* variable `a`. Binding `a` to `col: Colour # Empty` yields the extended *closed* row:

```
(xCoord: Int) # (yCoord: Int) # (col: Colour) # Empty
```

In this manner, when coupled with parametric polymorphism, extensible rows may simulate record subtyping [6].

A record calculus is *label polymorphic* if the same label name may label different types in different rows. For example, the rows:

```
(xCoord: Int) # Empty
(xCoord: Real) # (depth: Real) # Empty
(xCoord: a) # b
```

may all coexist within one program. As we shall see, the type system must work a little harder to ensure type correctness in the presence of polymorphic labels.

Rows are distinct from types, but may be used to form both record and variant types.

Records

A *record type* interprets a row as a product of label-indexed types. For example:

```
All ((xCoord: Int) # (yCoord: Int) # Empty)
```

is a record type with two labels. We write `All` to denote the record type constructor because records contain *all* elements of a row.

At the term level, we have the empty record `Triv` (of type `All Empty`), and a record extension operator `(l: _ && _)` for each label name `l`. (Throughout this paper we assume a distfix syntax for operators in which argument positions are written as `_`.) For example:

```
((xCoord: 1) && (yCoord: 2) && Triv)
```

is a record with the above record type.

Calculi typically also include a label selection operator `(_.l)` for each label name `l`. For our purposes we prefer to use pattern matching. For example:

```
let getYCoord = \((yCoord: z) && _) . z
in getYCoord ((xCoord: 1) && (yCoord: 2) && Triv)
```

evaluates to `2`.

Variants

Dually to records, a *variant type* interprets a row as a sum of label-indexed types. For example:

```
One ((isInt: Int) # (isBool: Bool) # Empty)
```

is a variant type with two labels. We write `One` to denote the variant type constructor because sums contain *one* element of a row.

At the term level, we have an injector `(Inj l: _)` for each label name `l`. For example:

```
(Inj isBool: True)
```

injects `True` with the label `isBool` into the above variant type. We also need a way to test a variant against a label. Again, for the purposes of this paper we will assume an injector may also be used as a pattern. For example:

```
case (Inj isBool: True) of {
  (Inj isBool: x) -> not x;
  otherwise -> True
}
```

evaluates to `False`.

(Notice the type `One Empty` contains only the undefined term.)

Soundness

Though liberal, record and variant calculi are not anarchic: somehow they must prevent a row from ever containing duplicate label names. For extensible record calculi this constraint requires some form of global analysis. For example, to reject (as surely we must) the program:

```
let f = \x y . ((xCoord: x) && y)
in (f 2 ((xCoord: 1) && Triv))
```

involves looking both at the definition and call sites for `f`.

A particularly elegant solution is to introduce qualified (constrained) polymorphism [12] and *insertion constraints* (called "lacks" constraints in [9].) We refer the reader to the work of Gaster and Jones [7] for a cogent exposition of this approach. Briefly, let-bound terms are assigned a type scheme which includes any constraints on the possible instantiations of quantified type variables. In the above example, `f` would be assigned the scheme:

```
forall a, b . xCoord ins b =>
  a -> All b -> All ((xCoord: a) # b)
```

which can be read as:

> "for all types `a` and rows `b` such that the label name `xCoord` may be inserted into `b`, the function from `a` and `All b` to `All ((xCoord: a) # b)`."

Now each use of `f` is free to instantiate `a` and `b`, but *subject to the constraint* `xCoord` *ins* `b`. Since our example program attempts to instantiate `b` to `((xCoord: Int) # Empty)`, which already contains the label name `xCoord`, it is rejected.

3 From Label- to Type-Indexed Rows

As a first step towards λ^{TIR}, consider naïvely erasing labels from the record and variant operators above.

We let the kind system keep rows, of kind `Row`, separate from types, of kind `Type`. Our presentation will be greatly simplified if we also allow *higher-kinds*, so that we may present our type operators as constants. We use : to denote "has kind" (and later, "has type.")

A *type indexed row* (TIR) is either the empty row or an extension of another row. Row extension is now free of label names:

```
Empty : Row
(_ # _) : Type -> Row -> Row
```

For example:

```
(Int # Bool # Empty)
```

is a closed row containing the element types `Int` and `Bool`. Rows are considered equal up to a permutation of their element types.

We also have two dual interpretations for a row: as a *type-indexed product* (TIP) or *type-indexed coproduct* (TIC) type:

```
(All _) : Row -> Type
(One _) : Row -> Type
```

A TIR is well-formed iff its element types are all distinct. Because we allow open rows, this cannot be verified locally, and so will be propagated using constraints. λ^{TIR} insertion constraints resemble those of record calculi, but with a type instead of a label. For example:

```
a ins (Int # Bool # Empty)
```

constrains `a` to be any type other than `Int` or `Bool`. Hence:

```
(List b) ins (Int # Bool # Empty)
```

is `true`: for every type `b`, `List b` cannot be equal to `Int` or `Bool`, and hence may be inserted into the row.

With the types and constraints in place, we now consider terms. A TIP is either the trivial product, or an extension of another:

```
Triv : All Empty
(_ && _) : forall (a : Type), (b : Row) .
             a ins b => a -> All b -> All (a # b)
```

A TIC is an injection of a term:

```
(Inj _) : forall (a : Type), (b : Row) .
            a ins b => a -> One (a # b)
```

Notice the use of insertion constraints to ensure the type `a` to insert does not already appear within the row `b` of the TIP or TIC.

For example:

```
(1 && True && Triv) : All (Int # Bool # Empty)
(Inj True) : forall (a : Row) . Bool ins a =>
               One (Bool # a)
```

We also allow any of the above three constants to appear within patterns. For example:

```
let flip = \(x && y && Triv) .
  ((1 - x) && (not y) && Triv)
in flip (True && 1 && Triv)
```

evaluates to `(0 && False && Triv)`. Notice the pattern `(x && y && Triv)` contains no explicit type information, and certainly no labels! It was the *type* of `x` within the body of `flip` which determined it was bound to `1` rather than `True`.

Case analysis of TIPs and TICs is possible using a *multi-way discriminator*, written as a brace-delimited sequence of λ-abstractions. For example, consider:

```
let flop = { \(Inj x) . 1 - x
           , \(Inj y) . if y then 0 else 1 }
in flop (Inj True)
```

The two λ-abstraction patterns will be tried in left-to-right sequence when `flop` is applied. In this case, the second pattern will match, and the term reduces to `0`. Since all functions in a discriminant must have the same type:

```
flop : forall (a : Row) .
         Int ins a, Bool ins a =>
           One (Int # Bool # a) -> Int
```

3.1 Equality Constraints

Consider a more challenging variation of the `flip` example:

```
let tuple = \(x && y && Triv) . (x, y)
in tuple (True && 1 && Triv)
```

(Here we assume λ^{TIR} to be enriched by conventional tuples, though they are easily encoded.) Unlike for `flip`, the body of `tuple` is fully polymorphic on the types of `x` and `y`. Hence:

```
tuple : forall (a : Type), (b : Type) .
          a ins (b # Empty) =>
            All (a # b # Empty) -> (a, b)
```

Now consider how to type-check the application of `tuple`. Assume its scheme has been specialised to fresh type variables `c` and `d`. Then we must unify rows `All (c # d # Empty)` and `All (Int # Bool # Empty)` subject to the constraint `c ins (d # Empty)`. Depending on which of `Int` or `Bool` we bind to `c`, this means the overall term has type `(Int, Bool)` or `(Bool, Int)`. Choosing one solution above another would destroy completeness of type inference. Rejecting such terms would prevent many useful examples (in particular, overloading: see Section 4).

Our solution is to introduce *equality constraints* to record which types and rows must be equal for a term to be well-typed. For example:

```
(c # d # Empty) eq (Int # Bool # Empty)
```

represents the constraint that `tuple` and its argument `(True && 1 && Triv)` agree in type. As with insertion constraints, equality constraints propagate until sufficient type information is available to simplify them.

For convenience, we allow equality constraints on both rows and types. (Type equality constraints may always be simplified down to row equality constraints as soon as they are introduced, hence they add no expressiveness to the system.)

Now consider:

```
let oneTrue =
  let tuple = \(x && y && Triv) . (x, y)
  in tuple (True && 1 && Triv)
in (1 - fst oneTrue, not (fst oneTrue))
```

Using equality constraints, we may assign `oneTrue` a *principal type scheme*:

```
forall (c : Type), (d : Type) .
  (c # d # Empty) eq (Int # Bool # Empty),
  c ins (d # Empty) =>
    (c, d)
```

Notice the first element of `oneTrue` has been used in both an `Int` context and `Bool` context, and the term reduces to `(0, False)`. To see how this works, consider each use of `oneTrue`. For the left use, `oneTrue` is specialised to a tuple with an `Int` first component. Hence its constraint is specialised to:

```
(Int # e # Empty) eq (Int # Bool # Empty),
Int ins (e # Empty)
```

where `e` is a fresh variable. This constraint may be simplified by binding `e` to `Bool`, and is thus `true`.

Similarly, for the right use the specialised constraint is:

```
(Bool # f # Empty) eq (Int # Bool # Empty),
Bool ins (f # Empty)
```

Again, the constraint is simplified to `true` with `f` bound to `Int`.

Membership and equality constraints interact in interesting ways. Indeed, much of the machinery of λ^{TIR} is devoted to the entailment and simplification of such mixed constraints. For example, the constraint:

```
Int ins (a # Empty),
(Int # Bool # Empty) eq (a # b # Empty)
```

may be simplified to `true` by binding `a` to `Bool` and `b` to `Int`. This is because the membership constraint prevents the binding of `a` to `Int`.

3.2 Newtypes

So far λ^{TIR} can only distinguish types *structurally*. In order to distinguish types by *name* we allow the programmer to introduce fresh type names, called (as in Haskell [20]) *newtypes*.

A newtype declaration takes the form:

$$\texttt{newtype}\ A = \backslash\Delta\ .\ \tau$$

where A is the newtype name, Δ a sequence of kinded type variables, and τ a type (of kind `Type`).

At the type level, newtype names behave as uninterpreted types (or, in general, *type constructors*). For example, assuming the declarations:

```
newtype A = \(a : Type) . a
newtype B = Int
newtype C = Int
```

then `A Int`, `A Bool`, `B`, `C` and `Int` are all distinct types.

At the term level, newtype names behave as single-argument data constructors. These may be used both to construct terms:

```
((A 1) && (A True) && (B 2) && (C 3) && 4) :
  All ((A Int) # (A Bool) # B # C # Int # Empty)
```

and to pattern match against terms in λ-abstractions:

```
\A x . x + 1 : A Int -> Int
\A x . not x : A Bool -> Bool
\B x . x + 1 : B -> Int
```

In effect, every newtype declaration introduces a polymorphic constant:

$$\boxed{A : \texttt{forall}\ \Delta\ .\ \tau \texttt{ -> } A\ \Delta}$$

Using newtypes, we can encode conventional monomorphic records by declaring a newtype for each label. For example, with declarations:

```
newtype xCoord = Int
newtype yCoord = Int
```

we have:

```
((xCoord 1) && (yCoord 2) && Triv) :
  All (xCoord # yCoord # Empty)
```

What about polymorphic record calculi? A obvious approach would be to declare each label to be the type-identity function:

```
newtype xCoord = \(a : Type) . a
newtype yCoord = \(a : Type) . a
```

This allows the newtypes to "label" terms of any type in any "record:"

```
((xCoord 1) && (yCoord 2) && Triv) :
  All ((xCoord Int) # (yCoord Int) # Empty)
((xCoord '1') && (yCoord "two") && Triv) :
  All ((xCoord Char) # (yCoord String) # Empty)
```

Unfortunately, it also allows the same newtype to appear within the same record, provided it labels terms of different types:

```
((xCoord 1) && (xCoord '1') && Triv) :
  All ((xCoord Int) # (xCoord Char) # Empty)
```

Though at first glance this may seem a useful generalisation of labels, we quickly run into problems when unifying rows containing them. For example, if `xCoord` really was a polymorphic label, then the following constraint should be simplified by by binding `a` to `Int`:

```
((xCoord a) # b) eq ((xCoord Int) # c),
(xCoord a) ins b,
(xCoord Int) ins c
```

However, as things stand, the simplifier would be *unsound* if it were to do so.

To see why, consider the possible substitution which binds `b` to `(xCoord Int) # Empty`, and `c` to `(xCoord Bool) # Empty`. The constraint becomes:

```
((xCoord a) # (xCoord Int) # Empty) eq
  ((xCoord Int) # (xCoord Bool) # Empty),
(xCoord a) ins ((xCoord Int) # Empty),
(xCoord Int) ins ((xCoord Bool) # Empty)
```

which implies `a` must be `Bool`, not `Int`. Hence, our simplifier is stymied by an excess of polymorphism.

Our solution is to introduce *opaque newtypes*, a variation of newtypes in which the type arguments are ignored when considering the simplification of insertion constraints.

Returning to our example, consider redeclaring the labels as:

```
newtype opaque xCoord = \(a : Type) . a
newtype opaque yCoord = \(a : Type) . a
```

Now the simplifier is free to bind `a` to `Int` in our constraint:

```
((xCoord a) # b) eq ((xCoord Int) # c),
(xCoord a) ins b,
(xCoord Int) ins c
```

This is because the membership constraint `(xCoord a) ins b` implies `b` cannot contain any type of the form `xCoord` τ, hence `b` cannot be extended to include `xCoord Int`, and hence `xCoord Int` must match `xCoord a`.

Furthermore, with `xCoord` declared as an opaque newtype, the term:

```
((xCoord 1) && (xCoord '1') && Triv)
```

is ill-typed:

```
error: the constraint
  (xCoord Int) ins ((xCoord Char) # Empty)
is unsatisfiable
```

Though at first glance they appear somewhat ad-hoc, opaque newtypes require very little special support within the machinery of λ^{TIR}.

3.3 Implementing Records

For the moment we put type-indexed rows aside and consider how to implement conventional label-indexed records. A naïve approach is as a map from labels to values, but then each access requires a dynamic lookup. A better approach, first suggested by Ohori [19], and independently, Jones [12], is to use the type information we already have to replace label names with natural number indices, and records with vectors. When a closed record is manipulated, these indices can be easily generated by finding a canonical ordering of label names. When an open record is manipulated within a polymorphic function, these indices must be passed as implicit arguments since their actual values will depend on how the function has been specialised.

This all seems rather complicated until it is noticed that indices propagate at run-time in parallel with insertion constraints at compile-time, except in the opposite direction.

Consider:

```
let f = \x . ((yCoord: 20) && x)
in f ((xCoord: 10) && Triv)
```

To ensure its body is well-formed, `f` is assigned the type scheme:

```
forall (b : Row) . yCoord ins b =>
  All b -> All ((yCoord: Int) # b)
```

At the application of `f`, `b` is specialised to `(xCoord: Int) # Empty`, and thus `f`'s constraint is specialised to `yCooord ins ((xCoord: Int) # Empty)`. This constraint is then introduced into the application's constraint context, where it may be simplified to `true`. Notice how `f`'s constraint propagated (at compile-time) from the site of its definition to the site of its use.

Now associate a run-time *index variable*, `w`, with `f`'s constraint `yCoord ins b`, with the understanding that `w` will be

bound at run-time to the *insertion index* of `yCoord` within whatever row `b` is specialised to. Or, to use OML's terminology [12], w will be bound to a *witness* of the satisfaction of the constraint that `yCoord` may be inserted into row `b`.

`f` is now compiled to a function accepting w as an additional implicit parameter:

$$\begin{array}{l}\mathsf{let\ f} = \lambda \mathsf{w}\ .\ \lambda \mathsf{x}\ .\ \mathsf{insert\ 20\ at\ w\ into\ x} \\ \mathsf{in}\ \ldots\end{array}$$

Here we use sans-serif font to denote run-time terms, and $\mathsf{insert}\ U\ \mathsf{at}\ W\ \mathsf{into}\ T$ inserts the term U at index W into the vector T.

In the application of `f`, again associate an index variable w' with the specialised constraint `yCooord` *ins* `((xCoord: Int) # Empty)`. This variable is passed to f, along with its argument:

$$\mathsf{f\ w'}\ \langle 10 \rangle$$

Here $\langle \ldots \rangle$ denotes a base-1 vector of run-time terms. (We shall use a special syntax for indices to prevent their semantic confusion with ordinary integers: One is the base index, and Inc W, Dec W the obvious offsets.)

Now when the simplifier rewrites `yCooord` *ins* `((xCoord: Int) # Empty)` to `true`, it is also obliged to supply a binding for w'. Assuming a lexicographic ordering on label names, `yCoord` should be inserted at index Inc One into the row `(xCoord: Int) # Empty`, hence w' is bound to the *absolute index* Inc One.

Thus the overall term is compiled as:

$$\begin{array}{l}\mathsf{let\ f} = \lambda \mathsf{w}\ .\ \lambda \mathsf{x}\ .\ \mathsf{insert\ 20\ at\ w\ into\ x} \\ \mathsf{in\ let\ w'} = \mathsf{Inc\ One} \\ \mathsf{in\ f\ w'}\ \langle 10 \rangle\end{array}$$

which reduces to the vector $\langle 10, 20 \rangle$.

Notice how the insertion index for `yCoord` within `b` was passed at run-time from the use site to the definition site, exactly in reverse of the propagation of the constraint `yCoord ins b` at compile-time.

This type-directed translation is an instance of the *dictionary-translation* [28]. We call a set of constraints with associated index variables a *constraint context*, by analogy with type contexts.

An index may sometimes depend on another. For example, the constraint context:

$$\mathsf{w} : \texttt{yCoord}\ \textit{ins}\ \texttt{((xCoord: Int) \# b)}, \mathsf{w'} : \texttt{yCoord}\ \textit{ins}\ \texttt{b},$$

can be simplified to `w : yCoord` *ins* `((xCoord: Int) # b)` by binding w' to the *relative index* Dec w. This is because `yCoord` will always be after `xCoord` in any row.

The same technique works for variants, which are represented as a pair of a natural number and value.

3.4 Implementing TIPs and TICs

Can we implement λ^{TIR} also using only natural number indices, vectors and pairs? The trick only works if we have an ordering on types. Clearly a total order on all types won't do, as then the relative ordering of non-ground types would depend on the *names* of their free type variables—disaster!

An obvious approach is to choose some ordering on monotypes, and only consider simplifying an insertion constraint υ *ins* τ_1 `#` $\ldots$ `#` τ_n `# Empty` when υ and each τ_i are ground. Then finding the index for υ is simply a matter of sorting these types. Unfortunately, since programs are often polymorphic all the way up to their top level, this means many insertion constraints will similarly propagate to the top level, leading to very large constraint contexts.

Thankfully, a less conservative ordering is possible. Assume we have a total order, $<^F$, on all built-in type constants (such as `Int`, `(All _)` and `(_ -> _)`), and all newtype names. Let $<^{F'}$ be $<^F$ extended to type variables, on which it is always false. So, for example:

$$\texttt{Int} <^{F'} \texttt{Bool} <^{F'} \texttt{String} <^{F'} \texttt{(\_ -> \_)} <^{F'} \ldots$$

but $\mathtt{a} \not<^{F'} \mathtt{Int}$ and $\mathtt{Int} \not<^{F'} \mathtt{a}$.

Every type τ has a *pre-order flattening*, denoted by $\mathit{preorder}(\tau)$. For example, $\mathit{preorder}(\texttt{A Int -> B Bool a}) = [\texttt{(\_ -> \_)}, \texttt{A}, \texttt{Int}, \texttt{B}, \texttt{Bool}, \texttt{a}]$. We then (roughly) define the partial order, $<$, on all types as follows:

$$\tau < \upsilon \iff \mathit{preorder}(\tau) <^{lex} \mathit{preorder}(\upsilon)$$

where $<^{lex}$ is the lexicographic ordering induced by $<^{F'}$. Notice that $<$ enjoys invariance under substitution, *viz*:

$$\tau < \upsilon \Longrightarrow \forall \theta\ .\ \theta\,\tau < \theta\,\upsilon$$

This property allows many insertion constraints to be discharged even when they contain type variables.

For example, consider the constraint:

$$\mathsf{w} : \texttt{(Bool -> a)}\ \textit{ins}\ \texttt{((Int -> b) \# Int \# Empty)}$$

All of these types may be totally ordered:

$$\texttt{Int} < \texttt{(Int -> b)} < \texttt{(Bool -> a)}$$

Thus we eliminate the constraint and bind w to Inc Inc One.

However, since the types in:

$$\mathsf{w} : \texttt{(Bool -> a)}\ \textit{ins}\ \texttt{((b -> c) \# Int \# Empty)}$$

cannot be totally ordered, this constraint cannot be further simplified.

The alert reader will notice we ignored the possible permutation of row elements in the above description. To account for this, we must first find the *canonical* order of every row within types before flattening them. The actual definition may be found in the full paper.

3.5 Ambiguity

λ^{TIR} type schemes sometimes quantify over type variables which appear only in the scheme's constraint. For example, in

```
forall a, b . (a # b) eq (Int # Bool # Empty) => a -> a
```

the variable `b` is not free in `a -> a`. However, since a binding for `a` uniquely determines a binding for `b`, this scheme is still sensible.

However, the scheme

```
forall a, b . b ins (Int # Bool # Empty) => a -> a
```

is inherently *ambiguous*. Since the insertion constraint may never be eliminated, it will float to the top-level of the program and cause an error. Furthermore, a binding for `b` cannot be chosen arbitrarily, since different bindings may lead to different indices, and hence change the behaviour of the program.

There are two implications of this. Firstly, an implementation of the simplifier should try to detect such obviously ambiguous constraints and report an error as soon as possible (much as for unsatisfiable constraints; see the next section). Secondly, the test for whether a programmer-annotated type scheme is an instance of a compiler-inferred type scheme is more complicated than the simple subsumption and entailment test used in other constraint-based systems. The actual definition may be found in the full paper.

3.6 Satisfiability

When a let-bound term is generalised, any residual constraints accumulated while inferring its type which mention quantified type variables are shifted into its type scheme. However, we would also like to be sure such constraints are *satisfiable*. This is for two reasons. Practically, it helps improve the locality of type error messages if unsatisfiable constraints are caught at the point of definition rather than at some remote point of use. Theoretically, it simplifies our proof of type soundness if every type scheme is known to have at least one satisfying instance.

Often, the simplifier will detect unsatisfiability in the course of examining each primitive constraint. For example, in:

```
let f = \x . ((xCoord 2) && (xCoord 1) && x)
in 1
```

assuming `x : All a`, then `f` has the constraint:

```
(xCoord Int) ins a,
(xCoord Int) ins ((xCoord Int) # a)
```

This will be simplified to `false`, which is easily detected when generalising.

However, sometimes the simplifier will fail to detect unsatisfiability, since it never speculatively unifies ambiguous rows. For example, in:

```
let g : All (Int # Bool # Empty) -> Int = ...
    h : All (Char # String # Empty) -> Int = ...
    f = \x y z . g (x && y && Triv) +
                 h (x && z && Triv)
in 1
```

assuming `x : a`, `y : b`, `z : c`, then `f` has the unsatisfiable constraint:

```
a ins (b # Empty), a ins (c # Empty),
(a # b # Empty) eq (Int # Bool # Empty),
(a # c # Empty) eq (Char # String # Empty)
```

Since this will not be further simplified to `false`, the system must explicitly test for satisfiability when generalising.

Unfortunately, this is not quite enough. Consider the example:

```
let f = \x . let g = \y . ((xCoord y) && x) in 1
in f ((xCoord 1) && Triv)
```

Assume `x : All a` and `y : b`. Then `g` has the satisfiable constraint:

```
(xCoord b) ins a
```

Thus `f` is assigned the type:

```
forall (a : Row) . All a -> Int
```

and the entire program has type `Int`.

However, under a naïve operational semantics for λ^{TIR}, β-reducing the application of `f` yields the program:

```
let g = \y . ((xCoord y) && (xCoord 1) && Triv)
in 1
```

Now `g`'s constraint becomes

```
(xCoord b) ins ((xCoord Int) # Empty)
```

which is unsatisfiable. Hence, subject-reduction fails for this semantics. (Our semantics will actually be denotational rather than operational, but the problem remains the same.)

This problem occurs only when a let-bound term is both unused *and* has a constraint mentioning type variables bound at an outer scope. In the above example, `g` was unused in the body of `f`, and `g`'s constraint contained the type variable `a` bound by `f`'s type scheme.

The simplest solution to this problem is to simply reject programs containing redundant let-bindings. Of course, an actual implementation would remove such bindings rather than reject the program. (Indeed, compilers tend to do this anyway as an optimisation.) This is the approach taken in OML [12, 13], and we adopt it for λ^{TIR}.

This approach works because if `x` is a let-bound variable with constraint C, and `x` is free in t, the satisfiability of t's constraint implies the satisfiability of C.

Now a constraint may be tested for satisfiability *regardless of the scope of its free type variables*. If the test fails, the constraint is unsatisfiable for any instantiation of outer-scope variables, and an error may be reported. If the test succeeds, no further processing is required, since the satisfiability test for any let-bound terms in an outer scope shall entail the satisfiability of the current constraint.

(Another solution to this problem is to introduce *existential constraints*. See the full paper for a discussion as to the pros and cons of this approach.)

4 An Example

This section presents a simple example of type-based overloading. Many more examples, including an encoding of XML-like "choice" and "unordered tuple" types, may be found in the full paper.

There are two approaches to overloading an identifier `x`. The *open-world* view, as adopted in Haskell's class system [28], assumes the multiple definitions for `x` are all instances of a common type scheme σ, but otherwise makes no assumptions about any particular definition. Hence, a new definition for `x` may be added without the need to recompile programs using `x`. This is most conveniently implemented by passing definitions as implicit parameters at *run-time* [12].

In contrast, the *closed-world* view, as adopted for method-overloading in Java [8] and many other object-oriented languages, assumes all definitions for `x` are known at each point of use, but otherwise only requires each definition to be *at a distinct type.* (Of course Java has a notion of subtyping which has no counterpart in λ^{TIR}, hence our examples are simpler.) Closed-world overloading is typically implemented by selecting the appropriate definition at *compile-time.* Hence, adding a new definition for `x` requires recompiling all programs using `x`, but there is no associated run-time cost.

λ^{TIR} is able to express cloded-world style overloading. In conjunction with *implicit parameters* [14], an open-world style of overloading is also possible, though unfortunately outside the scope of this paper.

For a classic example, assume we have two addition functions:

```
intPlus : Int -> Int -> Int
realPlus : Real -> Real -> Real
```

To overload `+` on both these definitions, we first build a TIP containing them:

```
let allPlus
  : All ((Int -> Int -> Int) #
         (Real -> Real -> Real) # Empty)
  = (intPlus && realPlus && Triv)
```

We then define `+` to project one element from `allPlus`:

```
let (+)
  : forall (a : Type), (b : Row) .
      a ins b,
      (a # b) eq
        ((Int -> Int -> Int) #
         (Real -> Real -> Real) # Empty) => a
  = (\(x && _) . x) allPlus
```

(This type scheme is actually inferred and need not be supplied by the programmer.)

Because `x` is used polymorphically in the λ-abstraction `\(x && _) . x`, the type inferencer cannot determine which of `Int -> Int -> Int` and `Real -> Real -> Real` should unify with its type `a`. Hence this equality constraint, and the membership constraint arising from the pattern `(x && _)`, must be deferred.

When typing the term

```
\y . (1 + 1, 1.0 + y)
```

we find it has type

```
e -> (c, f)
```

subject to the constraints introduced by each use of `+`:

```
(Int -> Int -> c) ins d,
((Int -> Int -> c) # d) eq
  ((Int -> Int -> Int) # (Real -> Real -> Real) # Empty),
(Real -> e -> f) ins g,
((Real -> e -> f) # g) eq
  ((Int -> Int -> Int) # (Real -> Real -> Real) # Empty)
```

The simplifier reduces this to `true`, with the bindings:

$$\begin{bmatrix} \texttt{c} \mapsto \texttt{Int}, \texttt{d} \mapsto \texttt{Real -> Real -> Real \# Empty}, \\ \texttt{e} \mapsto \texttt{Real}, \texttt{f} \mapsto \texttt{Real}, \texttt{g} \mapsto \texttt{Int -> Int -> Int \# Empty} \end{bmatrix}$$

Hence, the final inferred type is

```
Real -> (Int, Real)
```

However, for the term:

```
1.0 + 1
```

we find:

```
error: the constraint
  (Real -> Int -> a # b) eq
    ((Int -> Int -> Int) #
     (Real -> Real -> Real) # Empty)
is unsatisfiable
```

5 Type Checking

This section begins our formal development of λ^{TIR}. We'll introduce its syntax and kind system, and present the well-typing judgement. Well-typing requires the notions of constraint entailment, which in turn is built from a notion of type order. We conclude by demonstrating the soundness of our type system w.r.t. a simple denotational semantics.

Many details will be elided: please refer to the full paper for a more thorough exposition.

5.1 Syntax

Figure 1 presents the kinds, types and terms of the source language, most of which should be familiar from examples. Our presentation is made more uniform if we allow higher-kinds, type abstraction and type application, though care will be taken to avoid the need for higher-order unification. For simplicity the only base type is `Int`.

The empty constraint will be written as `true`, and a generic unsatisfiable constraint as `false`, though neither may appear explicitly within programs. We write $C +\!\!+ D$ to denote concatenation of the primitive constraints of C and D. Equality constraints are only allowed at kind `Type` or `Row`; we'll usually elide their annotation. As is customary, we identify the type scheme `forall` $\cdot$ `.` $\cdot$ `=>` τ with the type τ.

We identify the unitary discriminator `{ f }` with `f`. λ-abstractions contain patterns, which may be nested arbitrarily. We assume all pattern variables to be distinct, and will also assume no type or term variable binding ever shadows another.

Kinds	κ	::=	`Type` | `Row` | κ_1 `->` κ_2
Type variables	a, b	::=	`a`, `b`, ...
Newtype names	A, B	::=	`A`, `B`, ...
Types	τ, υ, ρ	::=	`Int` | υ `->` τ
		|	`Empty` | τ `#` ρ | `One` ρ | `All` ρ
		|	A | a | `\` a `:` κ `.` τ | τ υ
Row tails	l	::=	`Empty` | a
Type var context	Δ	::=	$a_1 : \kappa_1, \ldots, a_n : \kappa_n$ $\quad n \geq 0$
Prim constraints	c, d	::=	τ *ins* ρ
		|	τ *eq*$_\kappa$ υ $\quad \kappa \in \{$`Type`, `Row`$\}$
Constraints	C, D	::=	$c_1, \ldots, c_n$ $\quad n \geq 0$
Type schemes	σ	::=	`forall` Δ `.` C `=>` τ
Integers	i		
Variables	x, y, z	::=	`x`, `y`, `z`, ...
Abstractions	f	::=	`\` p `.` t
Terms	t, u	::=	i | A | `Inj` | t `&&` u | `Triv`
		|	t u | x | $\{f_1, \ldots, f_n\}$ $\quad n > 0$
		|	`let` x `=` u `in` t
Patterns	p, q	::=	i | A p | `Inj` p
		|	p `&&` q | `Triv` | x
Newtype decls	*tdecl*	::=	`newtype` $\{$`opaque`$\}^{opt}$ A `=` τ
Programs	*prog*	::=	$tdecl_1 \ldots tdecl_n$ t $\quad n \geq 0$

Figure 1: Syntax of λ^{TIR} kinds, types and terms.

In much of what follows we assume types and terms are represented in applicative form. For example, τ `->` υ is represented by the application `(_ -> _)` τ υ. Furthermore, we assume the binary operator `(_ # _)` to be generalised to a family of $(n+1)$-ary row consing operators `(#)`$_n$ for $n \geq 0$, so that τ_1 `#` $\ldots$ `#` τ_n `#` l may be represented by the single application `(#)`$_n$ $\tau_1 \ldots \tau_n$ l. (We also identify `(#)`$_0$ l with l.) We let F and G range over all constant type constructors and newtype names, and f and g range over the term constructors `(Inj _)`, `(Triv)` and `(_ && _)`.

We shall write $\overline{\tau}$ to denote $\tau_1 \ldots \tau_n$, and $\overline{\tau}_{\setminus i}$ to denote $\tau_1 \ldots \tau_{i-1}\ \tau_{i+1} \ldots \tau_n$. n will typically be clear from context.

λ^{TIR}'s type language forms a strongly normalising simply-typed λ-calculus with constants. We let Δ range over *kind-contexts* (mapping type variables to kinds), and let Δ_{init} denote the initial kind context containing the type constants defined in Section 3, and the names and kinds of all declared newtypes. We elide the rules of the *well-kinding* judgement $\Delta \vdash \tau : \kappa$, along with its extension to schemes, constraints and contexts. Both sides of an equality constraint must have the same kind; insertion constraints must be with a `Type` and a `Row`. Type schemes must have a body type of kind `Type`, and each universally quantified type variable must have kind `Row` or `Type`.

Every recursive newtype must be *well-founded*; *viz* every cycle passing through a newtype must also pass through at least one `All` or `One` type constructor.

Some judgements require *constraint contexts* in which every primitive constraint is associated with a unique index variable. $names(C)$ associates fresh witness names with each primitive constraint in C. $named(C)$ is the tuple of witness names of C. $anon(C)$ is C with all witness names removed. We write $norm(\tau)$ to denote the *β-normal form* for a type τ of kind `Type`. Newtype names are considered as free variables for the purpose of normalisation. $eqs(C)$ is the primitive equality constraints of C, and $inss(C)$ the primitive insertion constraints.

We let τ^m and υ^m range over all normalised monotypes of kind `Type` or `Row`.

Index vars	w	::=	$\mathsf{w}, \ldots$
Indices	W	::=	w | One | Inc W | Dec W | True
Bindings	B	::=	$w_1 = W_1, \ldots, w_n = W_n$ $\quad n \geq 0$
Variables	x, y, z	::=	$\mathsf{x}, \mathsf{y}, \mathsf{z}, \ldots$
Terms	T, U	::=	i | $\langle T_1, \ldots, T_n \rangle$ | Inj W T $\quad n \geq 0$
		|	$\lambda x\ .\ T$ | $\lambda(w_1, \ldots, w_n)\ .\ T$
		|	T U | $T\ (W_1, \ldots, W_n)$ | x | A | A^{-1}
		|	insert U at W into T | let $\langle\rangle = x$ in T
		|	let $x\ y =$ remove W from U in T
		|	case U of $\{$ Inj $W\ x \to T_1$; otherwise $\to T_2\}$
		|	case U of $\{i \to T_1$; otherwise $\to T_2\}$
		|	let $x = U$ in T | letw B in T

Figure 2: Syntax of run-time terms.

Figure 2 presents the syntax of the untyped run-time language—the target of our type-directed translation. Parts of this have already been introduced in Section 3.3.

TIP's are represented as ordered tuples $\langle T_1, \ldots, T_n \rangle$. TIC's are a pair Inj W T of an index and a run-time term. Each declared newtype `A` is represented by an injector A, and corresponding extractor A^{-1}. These are both the identity in any operational semantics, but will be needed within our model in Section 5.5.

We keep indices separate from run-time terms to simplify our soundness proof. One is the first index, and Inc W and Dec W offset index W by one position to the right or left. Indices are abstracted and passed in tuples, and may be let-bound by letw B in T. The "index" True witnesses the satisfaction of an equality constraint. It plays no part in an implementation, but makes the proofs of correctness more uniform.

let $\langle\rangle = U$ in T forces evaluation of U. In the term let $x\ y =$ remove W from U in T, x is bound to the term at index W in U, and y to the remaining product. The first case-form checks if U evaluates to a TIC with index W. The second simply checks for matching integers.

5.2 Well-typed Terms

We let Γ range over *type-contexts* (mapping variables to type schemes), and let Γ_{init} denote the initial type context containing the constants introduced in Section 3, and the newtype constructors introduced in Section 3.2.

Figure 3 presents the rules for deciding the *well-typing judgement* $\Delta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T$, with intended interpretation:

$$\boxed{\Delta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T}$$

$$\frac{\Delta \mid C \mid \Gamma \vdash t : v \texttt{ -> } \tau \hookrightarrow T \quad \Delta \mid C \mid \Gamma \vdash u : v' \hookrightarrow U \quad C \vdash^e v \; \mathit{eq}_{\mathtt{Type}} \; v' \hookrightarrow \mathsf{True}}{\Delta \mid C \mid \Gamma \vdash t \; u : \tau \hookrightarrow T \; U} \text{ APP}$$

$$\frac{\begin{array}{c}(x/f/A : \texttt{forall } \overline{a : \kappa} \texttt{ . } D \texttt{ => } \tau) \in \Gamma \\ D' = \mathit{named}(D) \quad \Delta \vdash \overline{v : \kappa} \\ C \vdash^e D'[\overline{a \mapsto v}] \hookrightarrow B\end{array}}{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash x/f/A : \tau[\overline{a \mapsto v}] \\ \hookrightarrow \mathsf{letw}\ B\ \mathsf{in}\ x/f/A\ \mathit{names}(D')\end{array}} \text{ VAR}$$

$$\frac{\Delta \mid C \mid \Gamma \vdash_1 f : v \texttt{ -> } \tau \hookrightarrow T[\bullet]}{\Delta \mid C \mid \Gamma \vdash \{f\} : v \texttt{ -> } \tau \hookrightarrow T[\mathsf{undefined}]} \text{ ABS}$$

$$\frac{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash_1 f_1 : \tau \hookrightarrow T[\bullet] \\ \Delta \mid C \mid \Gamma \vdash \{f_2, \ldots, f_{n+1}\} : \tau' \hookrightarrow U \\ C \vdash^e \tau \; \mathit{eq}_{\mathtt{Type}} \; \tau' \hookrightarrow \mathsf{True} \quad z \text{ fresh}\end{array}}{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash \{f_1, \ldots, f_{n+1}\} : \tau \\ \hookrightarrow \mathsf{let}\ z = U\ \mathsf{in}\ T[z]\end{array}} \text{ DISC}$$

$$\frac{\begin{array}{c}x \in \mathit{fv}(t) \quad \Delta \vdash D_1 \text{ constraint} \\ \Delta +\!\!+ \Delta' \vdash D_2 \text{ constraint} \\ C \vdash^e D_1 \hookrightarrow B \quad \mathit{saturate}(D_1 +\!\!+ D_2) \neq \emptyset \\ \Delta +\!\!+ \Delta' \mid D_1 +\!\!+ D_2 \mid \Gamma \vdash u : v \hookrightarrow U \\ \sigma = \texttt{forall } \Delta' \texttt{ . } \mathit{anon}(D_2) \texttt{ => } v \\ \Delta \mid C \mid \Gamma, x : \sigma \vdash t : \tau \hookrightarrow T\end{array}}{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash \texttt{let } x \texttt{ = } u \texttt{ in } t : \tau \\ \hookrightarrow \mathsf{let}\ x = (\mathsf{letw}\ B\ \mathsf{in}\ \lambda \mathit{names}(D_2)\ .\ U)\ \mathsf{in}\ T\end{array}} \text{ LET}$$

Figure 3: Well-typed λ^{TIR} terms. (The rule for integers has been elided.)

> "term t has type τ, and translates to the run-time term T, assuming the free term variables typed in Γ, the free type variables kinded in Δ, the satisfiability of the constraint context C, and the free index variables of C."

We intend the VAR rule to apply to variables (x), built-in constants (f), and newtypes (A).

Note that, as discussed in Section 3.6, the LET rule must check not only that the constraint for a let-bound term is well-kinded, but also that it is *satisfiable*, and that the let-bound variable appears free in the let body. The test for satisfiability uses the *saturate* function, which will be defined in Section 5.4.

Notice the symmetry of index abstraction in the LET rule and index application in the VAR rule.

The ABS and DISC rules both make use of the mutually recursively defined pattern compiler of Figure 4. n is the number of λ-abstractions of t to be compiled as patterns, and $T[\bullet]$ is the compiled run-time term with a "hole," $\bullet$, which should be filled by a term (of the same type) to evaluate should the pattern fail. The ABS rule fills the hole with undefined, since there is no other alternative to try. The DISC rule chains

$$\boxed{\Delta \mid C \mid \Gamma \vdash_n t : \tau \hookrightarrow T[\bullet]}$$

$$\frac{\Delta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T}{\Delta \mid C \mid \Gamma \vdash_0 t : \tau \hookrightarrow T} \text{ P1}$$

$$\frac{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash_{n+1} \backslash p\ .\ t : v \texttt{ -> } \tau \hookrightarrow T[\bullet] \\ \Delta \vdash \rho : \mathtt{Row} \quad C \vdash^e v \; \mathit{ins}\, \rho \hookrightarrow W \quad x, y \text{ fresh}\end{array}}{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash_{n+1} \backslash \mathtt{Inj}\ p\ .\ t : \mathtt{One}\ (v\ \#\ \rho) \texttt{ -> } \tau \\ \hookrightarrow \lambda x\ .\ \mathsf{case}\ x\ \mathsf{of}\ \{\mathsf{Inj}\ W\ y\ \to T[\lambda y\ .\ \bullet\ (\mathsf{Inj}\ W\ y)]\ y; \\ \mathsf{otherwise} \to \bullet\ x\}\end{array}} \text{ P4}$$

$$\frac{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash_{n+2} \backslash p\ .\ \backslash q\ .\ t : v_1 \texttt{ -> } v_2 \texttt{ -> } \tau \hookrightarrow T[\bullet] \\ C \vdash^e (\mathtt{All}\ \rho)\ \mathit{eq}_{\mathtt{Type}}\ v_2 \hookrightarrow \mathsf{True} \\ \Delta \vdash \rho : \mathtt{Row} \quad C \vdash^e v_1 \; \mathit{ins}\, \rho \hookrightarrow W \quad x, y, z \text{ fresh}\end{array}}{\begin{array}{c}\Delta \mid C \mid \Gamma \vdash_{n+1} \backslash p\ \&\&\ q\ .\ t : \mathtt{All}\ (v_1\ \#\ \rho) \texttt{ -> } \tau \\ \hookrightarrow \lambda x\ .\ \mathsf{let}\ y\ z = \mathsf{remove}\ W\ \mathsf{from}\ x \\ \qquad \mathsf{in}\ T[\lambda y\ .\ \lambda z\ .\ \bullet\ (\mathsf{insert}\ y\ \mathsf{at}\ W\ \mathsf{into}\ z)]\ y\ z\end{array}} \text{ P5}$$

$$\frac{\Delta \vdash v : \mathtt{Type} \quad \Delta \mid C \mid \Gamma, x : v \vdash_n t : \tau \hookrightarrow T[\bullet]}{\Delta \mid C \mid \Gamma \vdash_{n+1} \backslash x\ .\ t : v \texttt{ -> } \tau \hookrightarrow \lambda x\ .\ T[\bullet\ x]} \text{ P7}$$

Figure 4: Well-typed λ^{TIR} pattern abstractions. (The rules for integers, newtypes and `Triv` have been elided.)

together each discriminant such that failure of f_i will cause f_{i+1} to be tried.

Note than a "vanilla" λ-abstraction $\backslash x\ .\ t$ is typed by treating it as a singleton discriminator $\{\backslash x\ .\ t\}$ in the ABS rule. This in turn invokes the pattern rule P7 to remove the argument x, and then the rule P1 for the body t, which then continues in the well-typing judgement.

As a term is deconstructed, the pattern compiler must insert re-construction code so that failure will be handled correctly. A real compiler will attempt to β-reduce these once the hole has been filled.

At the heart of all these rules is the entailment judgement, $\vdash^e$, to be presented in Section 5.4. It is used in four ways:

(i) When two types must be equivalent (*e.g.*, in the APP and DISC rules) the type checker asks if the current constraint context entails their equality.

(ii) Whenever a row is constructed or pattern-matched (*e.g.*, in the P4 and P5 rules), the row must be well-formed (the insertion constraint satisfied), and an index must be known at run-time. The type checker thus asks if the current constraint context entails the membership constraint. If so, the entailment judgement yields the index W.

(iii) For technical reasons (covered in the full paper), the LET rule must allow the current constraint context to be weakened when checking a let-bound term. The type checker thus asks if the current constraint context entails the weakened constraint context.

(iv) Each variable occurrence propagates any constraints

from the variable's definition-site to the use-site. In the VAR rule, the type checker thus asks if the current constraint context entails the variable's constraints, suitably specialised.

We assume the following definitions for the source-language constants in the run-time language:

$$\begin{aligned}
\texttt{(Inj _)} &= \lambda(\mathsf{w}) \,.\, \lambda\mathsf{x} \,.\, \mathsf{Inj\ w\ x} \\
\texttt{(Triv)} &= \langle\rangle \\
\texttt{(_ \&\& _)} &= \lambda(\mathsf{w}) \,.\, \lambda\mathsf{x} \,.\, \lambda\mathsf{y} \,.\, \mathsf{insert\ x\ at\ w\ into\ y} \\
\texttt{A} &= \lambda\mathsf{x} \,.\, \mathsf{A\ x}
\end{aligned}$$

Notice their definitions match their types in Γ_{init}.

5.3 Type Order

Section 3.4 introduced the type ordering relation. In the full paper we define the binary function cmp_O such that if τ and υ are normalised types of kind `Type` or `Row`, then

(i) $cmp_O(\tau, \upsilon) = \mathbf{eq}$ if τ and υ are equal up to a permutation of row elements, and ignoring type arguments of newtypes in the set O;

(ii) $cmp_O(\tau, \upsilon) = \mathbf{lt}$ if τ is strictly less-than υ in the type ordering (and remains so under all substitutions);

(iii) $cmp_O(\tau, \upsilon) = \mathbf{gt}$ similarly; and

(iv) $cmp_O(\tau, \upsilon) = \mathbf{unk}$ if the relation between τ and υ is not invariant under all substitutions.

In practice, O will be instantiated to $\emptyset$ or *opaque*, the set of opaque newtypes of the program.

5.4 Constraint Entailment

Roughly speaking, a constraint C *entails* a constraint D, written $C \vdash^e D$, if every *satisfying substitution* for C satisfies every primitive constraint in D. However, we also ask that the satisfaction of each primitive constraint be *witnessed*. Hence the full judgement form is $C \vdash^e D \hookrightarrow B$, where B is a set of bindings of witness names of D to witnesses which may contain witness names from C. Thus B resembles a coercion from C to D.

5.4.1 Saturation

Our strategy for deciding entailment is to first *saturate* the equality constraints of C by reducing them to a set of unifying substitutions. We then discard those unifiers which violate any insertion constraints in C, and then check each primitive constraint in D is satisfied for each remaining unifier.

Figure 5 presents the definition for *saturate*. Much of the work is performed by $mgus_O$, which, given a set of equality constraints, collects the set of their most-general unifiers (if any). Here, "most-general" refers to the unifier for a fixed permutation of all rows, and does not imply *the set itself* is "most-general" in any sense. An empty unifier set implies a pair of types are un-unifiable. A non-singleton set implies at least one pair of rows are unifiable under more than one permutation of row elements.

$$\begin{aligned}
fv_O(a) &= \{a\} \\
fv_O(F\ \overline{\tau}) &= \mathbf{if}\ F \in O\ \mathbf{then}\ \emptyset \\
&\qquad \mathbf{else}\ \textstyle\bigcup_i fv_O(\tau_i) \\
fv_O(\texttt{(\#)}_n\ \overline{\tau}\ l) &= \textstyle\bigcup_i fv_O(\tau_i) \cup fv_O(l)
\end{aligned}$$

$$\begin{aligned}
mgus_O(\theta \vdash \texttt{true}) &= \{\theta\} \\
mgus_O(\theta \vdash b\ \texttt{eq}\ b, C) &= mgus_O(\theta \vdash C) \\
mgus_O(\theta \vdash b\ \texttt{eq}\ \tau, C) &= \mathbf{if}\ b \in fv_O(\tau)\ \mathbf{then}\ \emptyset \\
&\qquad \mathbf{else}\ mgus_O([b \mapsto \tau] \circ \theta \vdash \\
&\qquad\qquad C\ [b \mapsto \tau]) \\
mgus_O(\theta \vdash \tau\ \texttt{eq}\ b, C) &= mgus_O(\theta \vdash b\ \texttt{eq}\ \tau, C) \\
mgus_O(\theta \vdash F\ \overline{\tau}\ \texttt{eq}\ F\ \overline{\upsilon}, C) &= \mathbf{if}\ F \in O\ \mathbf{then}\ \{\theta\} \\
&\qquad \mathbf{else}\ mgus_O(\theta \vdash \overline{\tau\ \texttt{eq}\ \upsilon}, C) \\
mgus_O(\theta \vdash F\ \overline{\tau}\ \texttt{eq}\ G\ \overline{\upsilon}, C) &= \emptyset \qquad \text{when } F \neq G \\
mgus_O(\theta \vdash \texttt{(\#)}_m\ \overline{\tau}\ l\ \texttt{eq}\ \texttt{(\#)}_n\ \overline{\upsilon}\ l', C) &= \textstyle\bigcup_{1 \le j \le n} S_j \cup S' \\
&\text{where } S_j = \{mgus_O(\theta \vdash \tau_1\ \texttt{eq}\ \upsilon_j, \\
&\qquad \texttt{(\#)}_{m-1}\ \overline{\tau}_{\setminus 1}\ l\ \texttt{eq}\ \texttt{(\#)}_{n-1}\ \overline{\upsilon}_{\setminus j}\ l', C)\} \\
&\text{and } S' = \mathbf{if}\ l' = a\ \mathbf{and}\ a \notin fv_O(\tau_1)\ \mathbf{then} \\
&\qquad mgus_O([a \mapsto \tau_1\ \texttt{\#}\ b] \circ \theta \vdash \\
&\qquad\quad (\texttt{(\#)}_{m-1}\ \overline{\tau}_{\setminus 1}\ l\ \texttt{eq}\ \texttt{(\#)}_n\ \overline{\upsilon}\ b, C)[a \mapsto \tau_1\ \texttt{\#}\ b]) \\
&\qquad \mathbf{else}\ \emptyset \\
&\text{and } b : \texttt{Row}\ \mathsf{fresh}
\end{aligned}$$

$$\begin{aligned}
isIn(\tau, \texttt{(\#)}_n\ \overline{\upsilon}\ l) &= \exists i \,.\, cmp_{opaque}(\tau, \upsilon_i) = \mathbf{eq} \\
satisfied(C) &= \nexists(\tau\ \mathbf{ins}\ \rho) \in C \,.\, isIn(\tau, \rho) \\
saturate(C) &= \left\{ \theta \;\middle|\; \begin{array}{l} \theta \in mgus_\emptyset(\mathbf{Id} \vdash eqs(C)), \\ satisfied(\theta\ inss(C)) \end{array} \right\}
\end{aligned}$$

Figure 5: Definition of *saturate*

As in Section 5.3, O is a set of type constructors, and will be instantiated to either $\emptyset$ or (in the simplifier: see the full paper) *opaque*. For the later, the resulting "unifiers" need not unify the arguments of opaque newtypes.

Notice the case for row unification collects the unifiers for each possible matching of the first l.h.s. element type to each r.h.s. element type or the r.h.s. tail. Unifying a type with a row tail requires the introduction of a fresh type variable of kind `Row`, hence some care shall be required when stating properties involving $mgus_O$. Furthermore, no attempt is made to eliminate unifiers which lead to obviously ill-formed rows. For example

$$mgus_\emptyset(\mathbf{Id} \vdash \texttt{(Int \# Bool \# a)}\ \texttt{eq}\ \texttt{(String \# Int \# b)}) = \left\{ \begin{array}{l} [\texttt{a} \mapsto \texttt{String \# d}, \texttt{b} \mapsto \texttt{Bool \# d}], \\ [\texttt{a} \mapsto \texttt{String \# Int \# e}, \texttt{b} \mapsto \texttt{Int \# Bool \# e}] \end{array} \right\}$$

Here the second unifier (an instance of the first) duplicates the `Int` element types in both rows. This is in keeping with the definition of cmp_O. In the sequel we shall see how such unifiers are rejected when it comes to deciding entailment.

Furthermore, $mgus_O$ may also include "junk" unifiers which, though sound, are not most-general. For example

$$mugs_\emptyset(\mathbf{Id} \vdash \texttt{(a \# b \# Empty)}\ \texttt{eq}\ \texttt{(a \# b \# Empty)}) = \{\mathbf{Id}, \texttt{a} \mapsto \texttt{b}\}$$

$$\boxed{C \vdash^m \tau \; \mathit{ins} \; \rho \hookrightarrow W}$$

$$\frac{}{C \vdash^m \tau \; \mathit{ins} \; \mathsf{Empty} \hookrightarrow \mathsf{One}} \text{ MEMPTY}$$

$$\frac{(w : \tau' \; \mathit{ins} \; \rho') \in C \quad cmp_{opaque}(\tau, \tau') = \mathbf{eq} \quad cmp_{opaque}(\rho, \rho') = \mathbf{eq}}{C \vdash^m \tau \; \mathit{ins} \; \rho \hookrightarrow w} \text{ MREF}$$

$$\frac{cmp_{opaque}(\tau, \upsilon_i) = \mathbf{lt} \quad C \vdash^m \tau \; \mathit{ins} \; (\#)_n \; \overline{\upsilon} \; l \hookrightarrow W}{C \vdash^m \tau \; \mathit{ins} \; (\#)_{n-1} \; \overline{\upsilon}_{\setminus i} \; l \hookrightarrow W} \text{ MCONT}$$

$$\frac{cmp_{opaque}(\tau, \upsilon_i) = \mathbf{gt} \quad C \vdash^m \tau \; \mathit{ins} \; (\#)_n \; \overline{\upsilon} \; l \hookrightarrow W}{C \vdash^m \tau \; \mathit{ins} \; (\#)_{n-1} \; \overline{\upsilon}_{\setminus i} \; l \hookrightarrow \mathsf{Dec} \; W} \text{ MDEC}$$

$$\frac{cmp_{opaque}(\tau, \upsilon_i) = \mathbf{lt} \quad C \vdash^m \tau \; \mathit{ins} \; (\#)_{n-1} \; \overline{\upsilon}_{\setminus i} \; l \hookrightarrow W}{C \vdash^m \tau \; \mathit{ins} \; (\#)_n \; \overline{\upsilon} \; l \hookrightarrow W} \text{ MEXP}$$

$$\frac{cmp_{opaque}(\tau, \upsilon_i) = \mathbf{gt} \quad C \vdash^m \tau \; \mathit{ins} \; (\#)_{n-1} \; \overline{\upsilon}_{\setminus i} \; l \hookrightarrow W}{C \vdash^m \tau \; \mathit{ins} \; (\#)_n \; \overline{\upsilon} \; l \hookrightarrow \mathsf{Inc} \; W} \text{ MINC}$$

$$\boxed{C \vdash^e d \hookrightarrow W}$$

$$\frac{\forall \theta \in \mathit{saturate}(C) \; . \; cmp_{\emptyset}(\theta \; \tau, \theta \; \upsilon) = \mathbf{eq}}{C \vdash^e \tau \; \mathit{eq} \; \upsilon \hookrightarrow \mathsf{True}} \text{ EQUALS}$$

$$\frac{\forall \theta \in \mathit{saturate}(C) \; . \; \theta \; \mathit{inss}(C) \vdash^m \theta \; \tau \; \mathit{ins} \; \theta \; \rho \hookrightarrow W}{C \vdash^e \tau \; \mathit{ins} \; \rho \hookrightarrow W} \text{ INSERT}$$

$$\boxed{C \vdash^e D \hookrightarrow B}$$

$$\frac{C \vdash^e \overline{d \hookrightarrow W}}{C \vdash^e \overline{w : d} \hookrightarrow \overline{w = W}} \text{ CONJ}$$

Figure 6: λ^{TIR} constraint entailment

Here the second unifier is redundant, but to prevent its inclusion, or to detect and discard it, seems to be much more trouble than simply accounting for such unifiers in a few points within the correctness proofs.

Though we shall speak of *sets* of unifiers, *multi-sets* are also appropriate. Hence $mgus_O$ needn't attempt to collapse duplicate unifiers.

Of course an actual implementation of $mgus_O$ needn't use such a brute-force collection of all unifiers. By using a variation of cmp_O to first sort each row, many obviously failing combinations may be rejected.

5.4.2 Entailment Judgement

Figure 6 presents the constraint entailment judgements.

The rules of the ancillary judgement $C \vdash^m \tau \; \mathit{ins} \; \rho \hookrightarrow W$ attempt to find a suitable index, W, for type τ within row ρ. Notice these rules are non-deterministic: there may be many possible derivations, and hence many possible witnesses. Furthermore, infinite derivations are possible. Both these properties are an artifact of our presentation, which is pleasantly concise compared to a fully deterministic and finite system.

Rule MEMPTY is the obvious base case (recall indices are base 1). Rule MREF allows an index to be drawn from the environment, provided all types agree opaquely up to permutation. Notice that all comparisons in these rules use cmp_{opaque} rather than $cmp_{\emptyset}$, since the type arguments of opaque newtypes should not be significant in determining the insertion position of a type in a row.

The remaining rules all attempt to build a relative index by adding or removing a type from a row for which the index is known. This is only possible when the type being added or removed can be strictly ordered with respect to the type being inserted.

Sometimes W will be an absolute index. For example:

$$\mathtt{true} \vdash^m \mathtt{Bool} \; \mathit{ins} \; (\mathtt{Int} \; \mathtt{\#} \; \mathtt{String} \; \mathtt{\#} \; \mathtt{Empty}) \hookrightarrow \mathsf{Inc} \; \mathsf{One}$$

Otherwise, W will be relative to an index in C. For example:

$$\mathtt{w} : \mathtt{Bool} \; \mathit{ins} \; (\mathtt{a} \; \mathtt{\#} \; \mathtt{Empty}) \vdash^m \\ \mathtt{Bool} \; \mathit{ins} \; (\mathtt{a} \; \mathtt{\#} \; \mathtt{Int} \; \mathtt{\#} \; \mathtt{Empty}) \hookrightarrow \mathsf{Inc} \; \mathtt{w}$$

The rules for the $C \vdash^e d \hookrightarrow W$ judgement first saturate C, then check d is satisfied under each unifier. Notice that rule INSERT requires the index W witnessing $\tau \; \mathit{ins} \; \rho$ to be (syntactically) the same under each unifier. This prevents a membership constraint from being incorrectly discharged. For example, the following judgement is not true:

$$(\mathtt{a} \; \mathtt{\#} \; \mathtt{b} \; \mathtt{\#} \; \mathtt{Empty}) \; \mathit{eq} \; (\mathtt{Int} \; \mathtt{\#} \; \mathtt{String} \; \mathtt{\#} \; \mathtt{Empty}) \not\vdash^e \\ \mathtt{a} \; \mathit{ins} \; (\mathtt{Bool} \; \mathtt{\#} \; \mathtt{Empty}) \hookrightarrow W$$

Depending on whether `a` is bound to `Int` or `String`, W can be One or Inc One. When there are multiple ways to bind an index, we assume the entailment fails only if there is no single derivation which yields the same index under all unifiers. An actual implementation can avoid having to try many possible derivations of the $\vdash^m$ judgement by preferring relative to absolute indices.

Finally, the $C \vdash^e D \hookrightarrow B$ judgement extends the $C \vdash^e d \hookrightarrow W$ judgement from primitive constraints to full constraints. Notice this definition implies $\mathit{saturate}(C)$ is performed for each $d \in D$: of course an implementation needn't do so!

In the following, let $\mathit{sortingPerms}(\tau_1, \ldots, \tau_n)$ denote the set of all permutations $\pi : n \to n$ such that

$$\forall i, j \; . \; i < j \implies cmp_{opaque}(\tau_{\pi \, i}, \tau_{\pi \, j}) = \mathbf{lt}/\mathbf{eq}$$

Four properties of entailment are key:

(i) It is transitive and reflexive.

(ii) It is closed under substitution: $C \vdash^e D \hookrightarrow B$ implies $\theta \; C \vdash^e \theta \; D \hookrightarrow B$.

(iii) Types are tautologically equal only if they are equivalent: $\mathtt{true} \vdash^e \tau \; \mathit{eq} \; \upsilon$ implies $cmp_{\emptyset}(\tau, \upsilon) = \mathbf{eq}$.

$$[\![\texttt{All}\ (\texttt{(\#)}_n\ \overline{\tau^m}\ \texttt{Empty})]\!] =$$
$$\quad \mathbf{E}\ \{\mathsf{prod}_n : \langle v_1, \ldots, v_n\rangle \mid v_1 \in [\![\tau^m_{\pi\,1}]\!], \ldots, v_n \in [\![\tau^m_{\pi\,n}]\!]\}$$
$$[\![\texttt{One}\ (\texttt{(\#)}_n\ \overline{\tau^m}\ \texttt{Empty})]\!] =$$
$$\quad \mathbf{E}\ \{\mathsf{inj} : \langle i, v\rangle \mid 1 \le i \le n, v \in [\![\tau^m_{\pi\,i}]\!]\}$$
$$\quad \text{where } \pi \in \mathit{sortingPerms}(\tau^m_1, \ldots, \tau^m_n)$$

$$[\![\texttt{forall}\ \overline{a : \kappa}\ .\ C\ \texttt{=>}\ \tau]\!] =$$
$$\quad \bigcap \left\{ S_{(\overline{v^m}, B)} \;\middle|\; \begin{array}{l} \Delta_{init} \vdash \overline{v^m : \kappa}, \\ \texttt{true} \vdash^e D[\overline{a \mapsto v^m}] \hookrightarrow B \end{array} \right\}$$
$$\quad \text{where } D = \mathit{named}(C)$$
$$\quad \text{and } \mathit{names}(D) = (w_1, \ldots, w_n)$$
$$\quad \text{and } S_{(\overline{v^m}, B)} = \left\{ \mathsf{ifunc}_n : f \;\middle|\; \begin{array}{l} f \in \prod_{1 \le i \le n} \mathcal{I} \to \mathbf{E}\ \mathcal{V}, \\ f\ ([\![w_1]\!]_{env(B)}, \ldots, [\![w_n]\!]_{env(B)}) \\ \quad \in [\![\tau[\overline{a \mapsto v^m}]]\!] \end{array} \right\}$$

Figure 7: Denotation of λ^{TIR} normalized monotypes and type schemes as ideals of $\mathbf{E}\ \mathcal{V}$. (Extract)

(iv) A type may be tautologically inserted into a row only if the resulting row may be strictly ordered: $\texttt{true} \vdash^e \mathsf{w} : v\ \textit{ins}\ (\tau_1\ \texttt{\#}\ \ldots\ \texttt{\#}\ \tau_n\ \texttt{\#}\ \texttt{Empty}) \hookrightarrow B$ implies $\mathit{sortingPerms}(v, \tau_1, \ldots, \tau_n) = \{\pi\}$ (and thus v has index $\pi^{-1}\ 1$).

The definition of $\vdash^e$ is not *complete*, because it fails to exploit the lexicographic ordering of types. This may be redressed by including *projection* rules, however such rules appear to be both expensive and of uncertain benefit.

5.5 Soundness

The full paper presents a denotational call-by-name semantics for λ^{TIR} into the domain $\mathbf{E}\ \mathcal{V}$, where $\mathbf{E}$ is the usual lifting monad, and $\mathcal{V}$ the pre-domain given by

$$\begin{array}{l} \mathcal{V} = (\ (\mathsf{wrong} : \mathbf{1}) + (\mathsf{int} : \mathcal{Z}) + (\mathsf{func} : \mathbf{E}\ \mathcal{V} \to \mathbf{E}\ \mathcal{V}) \\ \qquad + (\textstyle\sum_{n \ge 0} \mathsf{prod}_n : \prod_{1 \le i \le n} \mathbf{E}\ \mathcal{V}) + (\mathsf{inj} : \mathcal{N}^+ \times \mathbf{E}\ \mathcal{V}) \\ \qquad + (\textstyle\sum_{n > 0} \mathsf{ifunc}_n : (\prod_{1 \le i \le n} \mathcal{I}) \to \mathbf{E}\ \mathcal{V})\ ) \end{array}$$

Here $+$ is categorical sum, $\to$ continuous (not necessarily strict) function space, $\mathbf{1}$ the set $\{*\}$, $\mathcal{Z}$ the set of integers, $\mathcal{N}^+$ the set of non-zero naturals, and $\mathcal{I}$ the set of indices. Each summand is tagged by a mnemonic for its injector. We use the summand $\mathsf{wrong} : *$ to denote all ill-typed programs.

Figure 7 presents an extract of the denotation of λ^{TIR} monotypes and type schemes as *ideals* [15] of $\mathbf{E}\ \mathcal{V}$. No monotype contains $\mathsf{wrong} : *$. Furthermore, with an ancillary lemma, we may also show that if the top-level constraint of a program is satisfiable, every scheme within it is also satisfiable. Hence every type scheme arising in a well-typed program is the intersection of ideals free of $\mathsf{wrong} : *$, and hence is also free of $\mathsf{wrong} : *$.

We have elided the denotation of terms, which is mostly straightforward.

We say η models Γ, written $\eta \models \Gamma$, if $\mathit{dom}(\eta) = \mathit{dom}(\Gamma)$ and for every $(x : \sigma) \in \Gamma$, $\eta\ x \in [\![\sigma]\!]$.

Theorem 1 (Soundness) If $\Delta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow T$, and θ is grounding and well-kinded under Δ, and $\texttt{true} \vdash^e \theta\ C \hookrightarrow B$, and $\eta \models \theta\ \Gamma$ then $[\![\mathsf{letw}\ B\ \mathsf{in}\ T]\!]_\eta \in [\![\theta\ \tau]\!]$.

6 Type Inference

In the full paper we show λ^{TIR} enjoys type inference.

We first define the *simplification* judgement $\langle \overline{a} \mid C\rangle \triangleright^* \langle \theta \mid D \mid B\rangle$. Here C is an input constraint, D a simplified output constraint, and θ a substitution such that $D \vdash^e \theta\ C \hookrightarrow B$ and $\theta\ C \vdash^e D$. Furthermore, C and D agree (up to the type variables in $\overline{a}$), on their ground instances.

Our constraint simplifier combines the notions of *context-improvement* and *context-simplification* developed by Jones [13] in an extension of his OML framework for qualified types [12].

Many of the row equality constraints which arise in typical programs can be eliminated by the simplifier in $O(n \log n)$ comparisons. All other row problems take $O(n^2)$ comparisons to consider (and possibly simplify.)

We then define the rules of the *type inference* judgement $\theta \mid C \mid \Gamma \vdash t : \tau \hookrightarrow B$. These follow closely those for type checking in Figures 3 and 4, except for two key differences. The first is that each primitive constraint tested for entailment during type checking must now be accumulated during type inference. The second is the inclusion of the simplification rule:

$$\frac{\begin{array}{c}\theta_1 \mid C' \mid \Gamma \vdash t : \tau \hookrightarrow T \\ \langle fv(\theta_1\ \Gamma) \cup fv_\emptyset(\tau) \mid C'\rangle \triangleright^* \langle \theta_2 \mid C \mid B\rangle\end{array}}{(\theta_2 \circ \theta_1)_{\restriction fv_\emptyset(\Gamma)} \mid C \mid \Gamma \vdash t : \theta_2\ \tau \hookrightarrow \mathsf{letw}\ B\ \mathsf{in}\ T}\ \textsc{simp}$$

This rule is used to simplify the current constraint context as often as required. In practice, it is applied after each derivation step.

At the time of writing we have not completed all cases of the proofs for soundness and completeness of type inference with respect to type checking. However, unlike in conventional type inference systems, λ^{TIR} type inference does not perform any type unification directly. Instead, it simply constructs the required equality constraints and relies on the above simplifier rule to do the actual work. Hence these proofs are routine (and tedious.) We have also yet to show coherence, which we find simplest to include with the statement of completeness.

7 Related Work

Record Calculi

Wand [30] first introduced *rows* to encode record subtyping (and, in turn, inheritance) using parametric polymorphism, though the system did not enjoy completeness of type inference. Remy [23] introduced label *presence* and *absence* flags in types, and demonstrated completeness of inference. Variations allowing *record concatenation* [9, 31] rather than just *record extension* were also proposed. Remy [22] has demonstrated concatenation may often be encoded using just extension.

Ohori [19] and, independently, Jones [12] developed polymorphic record and variant calculi, and a compilation method which represented records as natural-number indexed vectors. Ohori's system dealt only with closed rows; Jones' system allowed extensible rows. λ^{TIR} is a strict generalisation of Gaster and Jones' system of polymorphic extensible records [7]. The latter exploits *qualified types* and the *dictionary translation* [12] as a compilation method.

Parallel to the parametric polymorphism approach followed in this paper are record calculi based on *subtyping* [6].

Constrained Polymorphism

Odersky, Sulzmann *et al* have developed HM(X) [18] as a framework for constraint-based type inference. It adds to Jones' qualified types the notion of *constraint projection*, and guarantees any constraint domain X enjoying a *principle constraint* property can be lifted to a type-inference system enjoying completeness of type inference. Principle constraints are defined relative to a set S of constraints in *solved form*.

Since both Ohori's and Gaster and Jones' record calculi are instances of HM(X), we initially hoped λ^{TIR} would be likewise. Unfortunately, the definition of S for λ^{TIR} constraints appears to be as complicated as the definition of the constraint simplifier itself, and hence not particularly theoretically pleasing. Or, turning the picture around, our constraint simplifier is not *complete* for any particularly natural definition of S, as to do so would most likely be prohibitively expensive.

Our technical development is instead based upon Jones' more general framework for simplifying and improving qualified types [13]. Unfortunately, even that system proved to be too specialised to support λ^{TIR} directly. Hence, we have been forced to prove most of the correctness of our system from scratch.

Set Constraints

Set constraints are popular in program analysis [2, 1] and constraint logic programming [26]. λ^{TIR}'s constraint domain resembles a simple set-constraint domain with primitive subset constraints and set union. However, set-constraints have an *implicit* idempotency law:

$$a \cup \{b, b\} = a \cup \{b\}$$

whereas in λ^{TIR} this property is enforced by an *explicit* insertion constraint:

$$b \; \mathit{ins} \; a$$

Using this explicit form leads directly to our implementation method.

Despite this difference, it may still be possible to exploit some of the implementation techniques developed for set constraints if this proved necessary.

Intersection Types

λ^{TIR}'s type-indexed products bare a superficial resemblance to *intersection* types [21, 24]. (And coproducts to *union* types [3].) However, they differ fundamentally in their meaning, as λ^{TIR} products are not subject to any *coherency* condition w.r.t. a notion of subtyping. We explore this connection further in the full paper.

XML

XDuce [10] is another functional language with similar goals to XMλ, but built upon subtyping polymorphism. It has regular expressions as types, language containment as the subtyping relation, and does not depend on 1-nonambiguity. However, at the time of writing XDuce does not support parametric polymorphism or higher-order functions. Other proposals for XMλ-like languages build on regular-tree transducers [17] or existing functional languages [29].

8 Conclusions

Thanks to its notion of *type-indexed row*, and its expressive constraint domain of *insertion* and *equality* constraints, λ^{TIR} can naturally encode many programming idioms, including record calculi, anonymous sums and products, and closed-world style overloading. It can be straightforwardly compiled into an untyped run-time language in which type-indexing is reduced to conventional natural-number indexing. These indices are generated and passed at run-time as implicit arguments to let-bound expressions, exactly as occurs in some existing record calculi [19, 7].

For the programs we considered, the constraints were compact and reasonably intuitive. We are working on an implementation of λ^{TIR} within the larger language XMλ [16]. At the time of writing, our XMλ compiler can simplify constraints but not yet infer them. We hope to demonstrate the feasibility of λ^{TIR} on larger programs once this compiler is complete.

In common with most constraint-based type systems, λ^{TIR} constraints could conceivably grow to a size beyond the understanding of a programmer, and beyond the capability of the type inference system to solve. We anticipate that for typical programs this will not occur, however our hypothesis remains unverified until we can test it within XMλ.

Since entailment is incomplete, it is possible that a programmer-supplied type scheme may be an instance of an inferred scheme, but the system is unable to prove it. This may be redressed by adding projection rules to the $\vdash^m$ judgement to exploit the lexicographic ordering of types. However, we would like to gain some experience with the system before deciding if these potentially more expensive rules are justified.

On the theoretical side, we hope to complete a complexity analysis of constraint satisfiability, entailment and type inference as a whole. The latter may well be above **EXP**. However, as complexity class seems to be a poor indicator of the typical performance of type inference systems, our priority rests with completing the implementation.

Acknowledgements This paper has greatly benefited from the insightful comments of Alex Aiken, Simon Peyton Jones, Mark Jones, Martin Müller, John Launchbury, Daan Leijen,

Jeff Lewis and the POPL reviewers.

References

[1] A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35:79–111, 1999.

[2] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the ACM SIGPLAN Conference on Functional Programming Languages and Computer Architecture (FPCA'93), Copenhagen, Denmark*, pages 31–41, Jun 1993.

[3] F. Barbanera, M. Dezani-Ciancaglini, and U. de'Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, Jun 1995.

[4] A. Brüggemann-Klein and D. Wood. Unambiguous regular expressions and SGML document grammars. Technical Report 337, Computer Science Department, University of Western Ontario, London, Ontario, Canada, Nov 1992.

[5] P. Buneman and B. Pierce. Union types for semistructured data. In *Proceedings of the International Database Programming Languages Workshop (DBPL-7), Kinloch Rannoch, Scotland*, Sep 1999. Also available as University of Pennsylvania Dept. of CIS technical report MS-CIS-99-09.

[6] L. Cardelli and J. Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1(1):3–48, Mar 1991.

[7] B. R. Gaster and M. P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, Nov 1996.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series, Sun Microsystems. Addison-Wesley, 1996.

[9] R. Harper and B. Pierce. A record calculus based on symmetric concatenation. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL'91), Orlando, Florida*, pages 131–142, Jan 1991.

[10] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00), Montreal, Canada*, pages 11–22, Sep 2000.

[11] M. Jones and S. Peyton Jones. Lightweight extensible records for Haskell. In *Proceedings of the 1999 Haskell Workshop, Paris, France*, Oct 1999. Published in Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht.

[12] M. P. Jones. *Qualified Types: Theory and Practice*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[13] M. P. Jones. Simplifying and improving Qualified Types. In *Proceedings of the ACM SIGPLAN Conference on Functional Programming Languages and Computer Architecture (FPCA'95), La Jolla, California*, pages 160–169, Jun 1995.

[14] J. Lewis, M. Shields, E. Meijer, and J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00), Boston, Massachusetts*, pages 108–118, Jan 2000.

[15] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Programming Languages (POPL'84), Salt Lake City, Utah*, pages 165–174, 1984.

[16] E. Meijer and M. Shields. XMλ: A functional language for constructing and manipulating XML documents. (Unpublished. Draft available at `http://www.cse.ogi.edu/~mbs`), 1999.

[17] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *Proceedings of the Nineteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (SIGMOD/PODS 2000), Dallas, Texas*, pages 11–22, May 2000.

[18] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[19] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, Nov 1995. Earlier version appears in POPL'93, pp. 99–112.

[20] S. Peyton Jones and J. Hughes. *Haskell 98: A Non-strict, Purely Functional Language*, Feb 1999. Available at `http://www.haskell.org/onlinereport/`.

[21] B. C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, Apr 1997.

[22] D. Rémy. Typing record concatenation for free. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages (POPL'92), Albuquerque, New Mexico*, pages 166–176, Jan 1992.

[23] D. Rémy. Type inference for records in a natural extension of ML. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. The MIT Press, 1993. Earlier verision appears in POPL'89, pp. 77–87.

[24] J. C. Reynolds. Design of the programming language FORSYTHE. In P. W. O'Hearn and R. D. Tennent, editors, *ALGOL-like Languages*, pages 173–233. Birkhäuser, 1997.

[25] M. Shields. *Static Types for Dynamic Documents*. PhD thesis, Oregon Graduate Institute, Jan 2001. (To appear. Draft available at `http://www.cse.ogi.edu/~mbs`).

[26] F. Stolzenburg. An algorithm for general set unification and its complexity. *Journal of Automated Reasoning*, 22(1):45–63, Jan 1999.

[27] W3C. Extensible Markup Language, 2000. Available at `http://www.w3.org/XML/`.

[28] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages (POPL'89), Austin, Texas*, pages 60–76, Jan 1989.

[29] M. Wallace and C. Ranciman. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99), Paris, France*, pages 148–159, Sep 1999.

[30] M. Wand. Complete type inference for simple objects. In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science, (LICS'87), Ithaca, New York*, pages 37–44, Jun 1987. Corrigendum in LICS'88, p. 132.

[31] M. Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, Jul 1991.