CS 671, Automated Reasoning

Lesson 20: *Type Constructs based on Intersection (II): dependent records, abstract data types, basic algebra* April 3, 2001

Last time we discussed record types and their representation through intersection types. We had defined

$$\{x_1{:}T_1;\ ..\ ;\ x_n{:}T_n\} \equiv \{x_1{:}T_1\} \cap ..\ \cap \{x_n{:}T_n\}$$

where

$$\{x{:}T\} \equiv \texttt{z:Labels} \to \texttt{if z=}x\ \texttt{then}\ T\ \texttt{else Top}$$

That is, records are represented as *dependent functions* with a type of labels as domain whose range types depend on the value of the chosen label. Finite record type declarations are *underspecified type declarations* for these functions, as they only specify the type of the function's results for input labels that are explicitly mentioned. By intersecting these declarations we refine the function's range for other input labels as well and thus specify it for a larger collection of labels.[3]

We will now use the same methodology to define *dependent records*, i.e. types of the form

$$\{x_1{:}T_1;\ x_2{:}T_2[x_1];\ ..\ ;\ x_n{:}T_n[x_1;..x_{n-1}]\}$$

In a dependent record r, the type of a component x_i may depend on the values of the components $x_1..x_{i-1}$., that is $r.x_1{\in}T_1,\ r.x_2{\in}T_2[r.x_1],\ \ldots r.x_n{\in}T_n[r.x_1;..r.x_{n-1}]$.

This dependency is stronger than the dependencies that we have encountered in ordinary dependent records or dependent functions, as the type of the record r depends on the value of the record *itself* and not just on some external component.

Note, however, that the order of components in a record is not fixed. The above declaration is equal to $\{x_2{:}T_2[x_1];\ x_n{:}T_n[x_1;..x_{n-1}];\ ..;\ x_1{:}T_1\}$. Writing record type declarations in an order such that labels are "declared" before they are used in the type of some other label is only a matter of convention but, as we will see later, not necessary. Furthermore, the dependency may be mutual, that is we can even declare *mutually dependent records* like the following

$$\{x_1{:}T_1[x_2;..x_n];\ x_2{:}T_2[x_1;x_3;..x_n];\ ..\ ;\ x_n{:}T_n[x_1;..x_{n-1}]\}$$

It should be noted that the above notation only expresses a *possible* dependency. Usually, a component depends only on very few other components.

As usual, the canonical and noncanonical elements of the dependent type construct are the same as the ones of the independent ones. For records these are

$\{x_1{=}t_1;\ ..\ ;\ x_n{=}t_n\}$	finite record expression, $n{\geq}0$
$r.l$	component selection
$r.l{<}{-}t$	component assignment, also for new labels

[3] In an alternative definition $\{x{:}T\} \equiv \{x\} \to T$, where $\{x\} \equiv \{\texttt{z:Labels|z=}x \in \texttt{Labels}\}$, the nature of the underspecification is an "incomplete" domain description. As functions in type theory are generally polymorphic, $\{x\} \to T$ describes the class of all functions, which on inputs in $\{x\}$ return results in T (while in set theory $\{x\} \to T$ is the class of functions that are only defined on $\{x\}$). By intersecting such declarations we extend the function's domain description to a larger collection of labels, i.e. specify its behavior more precisely on that domain. The overall effect is the same.

Formalizing dependent records has been an open research problem for many years, as they were known to have a variety of useful applications in mathematics and programming that cannot be expressed elegantly with other type constructs. In particular they make it possible to develop a formal account of abstract data types and objects in programming and to build a hierarchy of mathematical concepts. Let us look at two common examples.

Mathematics: Algebra

A *semigroup* is a tuple $(M, \circ)$ where M is a carrier type and $\circ{:}M\times M\to M$ is associative. In principle, it is possible to define semigroups via dependent products

$$\texttt{SemiGroup} \equiv \texttt{M:}\mathbb{U} \;\times\; \circ\texttt{:M}\times\texttt{M}\to\texttt{M} \;\times\; \forall\texttt{x,y,z:M. x}\circ\texttt{(y}\circ\texttt{z) = (x}\circ\texttt{y)}\circ\texttt{z} \in \texttt{M}$$

However, this definition actually defines semigroups to be triples $(M, \circ, pf)$, where pf is a proof expression showing the associativity of $\circ$, which is not what we had in mind. A possible solution for that problem is to use the subset constructor, which we will introduce in one of the coming lectures, and to define

$$\texttt{SemiGroup} \equiv \{\ \texttt{sg:SemiGroupSig}\ |\ \forall\texttt{x,y,z:M}_{sg}.\ \ \texttt{x}\circ_{sg}\texttt{(y}\circ_{sg}\texttt{z) = (x}\circ_{sg}\texttt{y)}\circ_{sg}\texttt{z} \in \texttt{M}_{sg}\ \}$$

where `SemiGroupSig` denotes the *signature* of semigroups, which again would be defined as a dependent product. While this definition gives us the correct kind of objects, it also shows how tedious it is to access their components in a formal notation.

Furthermore, the use of dependent records makes it difficult to describe extensions of semigroups such as monoids or groups in a natural way. A *monoid* is a triple $(M, \circ, e)$, where $(M, \circ)$ is a semigroup and $e{:}M$ is an identity wrt. $\circ$. Similarly, a *group* is a quadruple $(M, \circ, e, ^{-1})$, where $(M, \circ, e)$ is a monoid and $^{-1}{:}M\to M$ is an inverse function wrt. $\circ$.

Intuitively, monoids and groups are considered *special instances* of semigroups that are created simply by adding new components and axioms. However, if these concepts are represented by dependent products, they cannot be related them via subtyping, which would be the natural way to express refinement of concepts. The relation

$$\begin{array}{lll} & \texttt{GroupSig} & = \texttt{M:}\mathbb{U} \times \circ\texttt{:M}\times\texttt{M}\to\texttt{M} \times \texttt{e:M} \times {}^{-1}\texttt{:M}\to\texttt{M} \\ \sqsubseteq & \texttt{MonoidSig} & = \texttt{M:}\mathbb{U} \times \circ\texttt{:M}\times\texttt{M}\to\texttt{M} \times \texttt{e:M} \\ \sqsubseteq & \texttt{SemiGroupSig} & = \texttt{M:}\mathbb{U} \times \circ\texttt{:M}\times\texttt{M}\to\texttt{M} \end{array}$$

does not hold because of the structural differences between the three types.

Dependent records, on the other hand, allow us to express these concepts in a way that respects the natural refinement relation between them. If we define

$$\begin{array}{ll} \texttt{SemiGroupSig} & \equiv \{\texttt{M:}\mathbb{U};\ \circ\texttt{:M}\times\texttt{M}\to\texttt{M}\} \\ \texttt{SemiGroup} & \equiv \{\texttt{M:}\mathbb{U};\ \circ\texttt{:M}\times\texttt{M}\to\texttt{M};\ \texttt{assoc:}\ \ \forall\texttt{x,y,z:M. x}\circ\texttt{(y}\circ\texttt{z) = (x}\circ\texttt{y)}\circ\texttt{z} \in \texttt{M}\} \end{array}$$

then `SemiGroup` $\sqsubseteq$ `SemiGroupSig` is a trivial consequence of one of the basic properties of records, namely $\{x{:}S;\ y{:}T[x]\} \sqsubseteq \{x{:}S\}$.

Furthermore, it is even possible to express the fact, that monoids and groups are created from semigroups by some form of inheritance. For instance, we may define

$$\begin{array}{ll} \texttt{MonoidSig} & \equiv \{\texttt{SemiGroupSig; e:M}\} \\ \texttt{GroupSig} & \equiv \{\texttt{MonoidSig;}\ {}^{-1}\texttt{:M}\to\texttt{M}\} \end{array}$$

where $\{R_1;R_2\}$ expresses record type concatenation (with implicit suppression of braces).

Note that in both cases the name `M` refers to a label that is assumed to be declared in `SemiGroupSig` and `MonoidSig`.[4]

With the above definitions, we may even define groups and monoids through *multiple inheritance* as follows

```
Monoid  ≡ {SemiGroup; MonoidSig; id:  ∀x:M. e∘x=x ∈ M}
Group   ≡ {Monoid;    GroupSig; inv:  ∀x:M.x∘x⁻¹=e ∈ M}
```

Note, that the labels `M` and ∘ occur in both `SemiGroup` and `MonoidSig`. However, this does not create a problem since record declarations allow multiple occurrences of the same label – the corresponding record component simply has to be a member of all the types mentioned.

The diagram to the right shows the refinement hierarchy between the six concepts (read left-to-right, bottom-to-top), which formally follows directly from the above definitions.

$$\begin{array}{lcl} \texttt{SemiGroup} & \sqsubseteq & \texttt{SemiGroupSig} \\ \sqsubseteq & & \sqsubseteq \\ \texttt{Monoid} & \sqsubseteq & \texttt{MonoidSig} \\ \sqsubseteq & & \sqsubseteq \\ \texttt{Group} & \sqsubseteq & \texttt{GroupSig} \end{array}$$

Note that the above definitions still require semigroups, monoids, and groups to have proof components for the corresponding axioms. This can be avoided if we allow records to contain *squashed components*. The squash operator is defined using set types that we will introduce in one of the upcoming classes.

$[P] \equiv \{\texttt{x:Top} \mid P\}$

$[P]$ is equal to `Top` if P has a proof (i.e. is true) and is empty if P is false. By squashing a proposition, we remove its computational content without removing the requirement to prove it. In record types, assigning a squashed type $[P]$ to a component allows the corresponding to assign anything to this component. However, if we want to prove membership, we have to prove P in the process. These considerations lead to the following definitions.

```
SemiGroup ≡ { SemiGroupSig;      assoc:  [∀x,y,z:M. x∘(y∘z) = (x∘y)∘z ∈ M] }
Monoid    ≡ { SemiGroup; MonoidSig; id:  [∀x:M. e∘x=x ∈ M]}
Group     ≡ { Monoid;    GroupSig; inv:  [∀x:M.x∘x⁻¹=e ∈ M]}
```

Programming: Abstract Data Types

Typically, abstract data types are defined by introducing a collection of names for data types, a type declaratiuons for newly defined operator names, and a collection of axioms. The type declarations usually involve the names of the newly introduced data types and the axioms involve both the types and the newly defined operator names.

An abstract data type `STACK`(T) of stacks over some type T, for instance, can be declared as follows

[4] There are no syntactical requirements for `M` to be "bound", as `M` is a label and not a free variable. However, a well-formedness proof of `MonoidSig` requires `M` to be declared as a type instead of just being an object in `Top`.

```
TYPES                 Stack
OPERATORS             empty:   Stack
                      push:    Stack×T → Stack
                      pop:     {s:Stack|s≠empty} → Stack×T
AXIOMS                pushpop: ∀s:Stack.∀t:T.  pop(push(s,a)) = (s,a)
```

Again, dependent products could be used to represent abstract data types. For instance, one might define

```
STACK(T) ≡    Stack:𝕌
            × empty:   Stack
            × push:    Stack×T → Stack
            × pop:     {s:Stack|s≠empty} → Stack×T
            × pushpop: ∀s:Stack.∀t:T.  pop(push(s,a)) = (s,a)
```

but as before, this definition does not allow relating two abstract data types through subtyping, nor does it support the extension of abstract data types through inheritance. Furthermore, an implementation of the abstract data type would not only require the four components to be provided, but also a proof object for the axiom `pushpop`.

With dependent records, all these issues can be expressed straightforwardly. If we define

```
STACKSIG(T) ≡ { Stack:𝕌
              ; empty:   Stack
              ; push:    Stack×T → Stack
              ; pop:     {s:Stack|s≠empty} → Stack×T
              }
```

then

```
STACK(T) ≡ {STACKSIG(T); pushpop: [∀s:Stack.∀t:T.pop(push(s,a))=(s,a)∈M]}
```

and the relation `STACK(T)` ⊑ `STACKSIG(T)` follows immediately. Since we squashed the axioms, we can provide elements of `STACKSIG(T)` as implementations of `STACK(T)` such as the following implementation of stacks through lists

```
list-as-stack(T) ≡ { Stack = T list
                   ; empty = []
                   ; push  = λs,t.  t::s
                   ; pop   = λs. <hd(s),tl(s)>
                   }
```

and then prove that they satisfy the `pushpop` axiom.

```
⊢ list-as-stack(T) ∈ STACK(T)
```

Representing Dependent Records in Type Theory

Representing dependent records in Type Theory requires us to use some form of self-reference, since a dependent record $r \in \{x{:}S;\ y{:}T[x]\}$ has to satisfy $r.y \in T[r.x]$.

In an earlier approach [Hic96], Jason Hickey introduced the concept of *very dependent function types*, which extends the type of dependent functions $x{:}S \to T[x]$ by a self-dependeny requirement: a function f has to be an element of the type $x{:}S \to T[f,x]$.

While this type turned out to be sufficient in theory, it led to very complex formal proofs: wellformedness goals require proving that $T[f, a]$ only depends on values $b < a$ for some well-founded order relation $<$ on S.

Using dependent intersection types leads to a much simpler solution [Kop00]. In fact, building dependent record types from singleton records can be done in the same way as independent record types are build using binary intersection. For these types we had defined

$$\{x{:}S;\ y{:}T\} \quad\equiv\quad \{x{:}S\} \cap \{y{:}T\}$$

which makes sure that $r.x \in S$ and $r.y \in T$ for $r \in \{x{:}S;\ y{:}T\}$. For a dependent record $r \in \{x{:}S;\ y{:}T[x]\}$ the condition $r.x \in S$ is the same as before but now $r.y \in T[r.x]$ must hold. This can naturally be expressed by the definition

$$\{x{:}S;\ y{:}T[x]\} \quad\equiv\quad \mathtt{r}{:}\{x{:}S\} \cap \{y{:}T[r.x]\}$$

Recall that $\{x{:}S\}$ and $\{y{:}T[r.x]\}$ are underspecified declarations of functions, the first requiring that on input $x \in$ `Labels` the function r must return a value of type S, while the second requires that on input $y \in$ `Labels` it must return a value of type $T[r.x]$. The intersection of these two declarations is exactly what dependent records express.

Below we summarize all the definitions that are necessary for building (dependent) record types and operations of records

$$\begin{array}{lcl}
\{x{:}T\} & \equiv & \mathtt{z{:}Labels} \to \mathtt{if\ z{=}}x\ \mathtt{then}\ T\ \mathtt{else\ Top} \\
\{R_1;\ R_2\} & \equiv & R_1 \cap R_2 \\
\{x{:}S;\ y{:}T[x]\} & \equiv & \mathtt{r}{:}\{x{:}S\} \cap \{y{:}T[r.x]\} \\
 & & \\
r.l & \equiv & (r\ l) \\
r.l\mathtt{<-}t & \equiv & \lambda\mathtt{z.}\ \ \mathtt{if\ z{=}}l\ \mathtt{then}\ t\ \mathtt{else}\ r\mathtt{.z} \\
\{\} & \equiv & \lambda\mathtt{l.()} \\
\{r;\ l{=}t\} & \equiv & r.l\mathtt{<-}t
\end{array}$$

The last definition is only syntactical sugar for building record expressions. Together with a display form for iteration it leads to

$$\{x_1{=}t_1;\ \ldots\ ;\ x_n{=}t_n\} \equiv \{\}.x_1\mathtt{<-}t_1.\ \ \ldots\ \ \ x_n\mathtt{<-}t_n$$

Discussion

Dependent intersection types enforce a certain order in the presentation of dependent records. They do not allow labels to occur before they are declared. As a consequence, mutually dependent record types cannot be defined via intersection types as well. It is very likely that mutually dependent record types can be expressed using very dependent function types, but the corrsponding well-formedness issues still needs to be explored.

In most application examples, the order of certain record labels should be irrelevant. In abstract data types, for instance, there are only dependencies between the declared types and the declarations of the operations, and dependencies between these two and the axioms. This enables us to define new abstract data types through inheritance. For instance, one might extend the data type of stacks by a selection function

$\texttt{STACKSEL(}T\texttt{)} \equiv \{\texttt{STACK(}T\texttt{); select: } \{\texttt{s:Stack|s}\neq\texttt{empty}\}\times\mathbb{N} \rightarrow T\}$

On the other hand, a direct definition of STACKSEL(T) would usually be written as follows.

```
STACKSEL(T) ≡ { Stack:𝕌
              ; empty:   Stack
              ; push:    Stack×T → Stack
              ; pop:     {s:Stack|s≠empty} → Stack×T
              ; select:  {s:Stack|s≠empty}×ℕ → T
              ; pushpop: [∀s:Stack.∀t:T.  pop(push(s,a)) = (s,a) ∈ M]
              }
```

As types, these two definitions are *not equal*, since the order matters for the semantical equality of types. However, it is possible to prove mutual refinement, which means that both types have the *same elements*.

A formal proof of this property requires us to prove that in certain cases the order of labels in a record may be swapped, i.e. that

$$\vdash \{x_1{:}T_1;\ x_2{:}T_2\} \doteq \{x_2{:}T_2;\ x_1{:}T_1\}$$

whenever the labels x_1 and x_2 do not occur in T_1 and T_2 (where $S \doteq T \equiv S \sqsubseteq T \wedge T \sqsubseteq S$). Testing this condition requires a *syntactical* check, which becomes even more difficult since x_1 and x_2 are labels and not free variables.

While it is possible to write tactics that prove two record types "equal" (wrt. $\doteq$) by swapping the order of labels and proving that each swap operation preseverves $\doteq$, it would be more desirable to state a theorem of the form

> *Whenever two record types are syntactically equal up to reordering of labels then they are semantically equal wrt.* $\doteq$

which would allow us to prove the equality of records without having to *execute* the "swap"-tactic. This form of reasoning, however, requires *meta-reasoning* and *reflection*, which we will discuss in one of the upcoming lectures.

References

[Hic96] Jason Hickey. Formal objects in type theory using very dependent types. In *Foundations of Object Oriented Languages 3*. Williams College, 1996.

[Kop00] Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. Technical Report TR 2000-1809, Cornell University. Department of Computer Science, 2000.

Remarks:

At this point, dependent intersection types (and union types) are not yet implemented in Nuprl but only in the MetaPRL proof environment.