Why We Can't have SML Style `datatype` Declarations in HOL

Elsa L. Gunter

AT&T Bell Laboratories, Rm. #2A-432, Murray Hill, NJ, 07974-0636, USA

Abstract

The type descriptions that `define_type` is capable of handling are noticeably more limited than those allowed by SML. In particular, `define_type` requires of a type description it is given that the type being defined should not occur within any compound type. While this restriction is more severe than is necessary for there to be a solution in HOL to the description, we show that some restriction on the nature of the compound types within which the type being defined may occur is necessary. Not all descriptions allowable in SML will have a solution in HOL. Moreover, owing to the nature of the basic principle of type definition in HOL, no purely syntactic non-ad hoc test of a recursive type description will be sufficient to allow us to extend `define_type` to compound types involving "safe" type constructors such as `list` while at the same time barring all descriptions for which no solution is possible. Any general extension to the `define_type` package that allows the types being defined to occur within compound types in the type description will need to take as additional arguments theorems about the type constructors used in the compound type that justify their being so used. Finally, we show that an extension to the case where all the type constructors used in compound types involving the types being defined are essentially recursive type constructors themselves; the type constructors must satisfy an "initiality" theorem of the form returned by `define_type`, which must be supplied as an argument to this extension of `define_type`.

Keyword Codes: F.4.1; I.2.3
Keywords: Mathematical Logic; Deduction and Theorem Proving.

1 Introduction

In the early days of HOL, people using the system rarely made new types. This was considered to be in the domain of only the most sophisticated of user, and among them it was left to those who were working on extending the basic system (*e.g.* adding the type of lists or integers). With the advent of Tom Melham's `define_type` package, this situation has changed dramatically. Now that it is roughly as easy to define new types, acquire properties of elements of those types and define new functions over those types as to create a `datatype` declaration in SML, it has become a commonplace aspect of setting up a new problem in HOL to create several types via `define_type` to ease the process. But

the type descriptions that `define_type` is capable of handling are noticeably more limited than those allowed by SML. In particular, `define_type` requires of a type description it is given that the type being defined should not occur within any compound type. As people have become more accustomed to using `define_type` to create new types to solve their problems, they have come to want the greater flexibility that would come from allowing the type being defined to occur within a compound type in its description. The question has been asked "Why can't I do the following:

```
define_type "x1" "x1 = A1 | B1 ((x1)list)";
```

Is there some intrinsic reason for the limitation which forbids this?" In this paper we intend to show that in fact there is an intrinsic reason why we must have some such limit. The particular description given above, using the type constructor `list` in fact causes no problem. However, we intend to show that there is no syntactic test, shy of an ad hoc enumeration of some favorite "safe" type constructors, that will allow us to be able to determine if it is possible to define in HOL a type allowing the appropriate principle of recursive function definition. That is, while we could explicitly extend `define_type` to handle `list`, `pair` and a few others, in general it will not be possible to generically construct a solution for a description such as

```
define_type "x1" "x1 = A1 | B1 ((x1)foo)";
```

or even to determine if one exists.

2 Differences Between SML Types and HOL Types

Although the languages of SML and HOL look very similar, there are several important differences that HOL users should be aware of. One difference of which most users are aware is that every HOL type must have an element of that type, whereas it is possible to define a type in SML which has no elements. The SML declaration

```
datatype Empty = Nothing of Empty
```

causes no problems, whereas the HOL command

```
define_type "Empty = Nothing Empty"
```

is rejected because of the lack of a non-recursive operator.

There is another significant way in which the types of SML and HOL differ, and that is with regard to the meanings of function spaces. In SML the function space between two types is taken to mean the "computable functions" between two domains which are the meanings of the two types. There are various ways of defining "domain" and "computable", but it is important to note that we cannot use sets for domains and the full set of functions between them for the meaning. On the other hand, we can with HOL; HOL admits a set-theoretic model. Since we can interpret types as sets and functions as all functions between the sets, we cannot have a solution, for example, to the description

```
"lambda = Var num | App lambda lambda | Abs (lambda -> lambda)"
```

For a type to be a solution to this description would require that there be a one-to-one mapping `Abs` of the function space over the type into itself. Moreover, there would need to be a one-to-one mapping `Var` of the naturals into it as well. However, given a set A, the set $A \to A$ can be injected into it if and only if A is a singleton set. For any set A of cardinality greater than 1, the cardinality of $A \to A$ is always strictly greater than that of A. This is usually shown using the Cantor Diagonal argument, but can also be shown by a variation of Russell's Paradox. The reason this problem does not arise in SML is because the meaning of the function space is necessarily not the full function space between sets, but rather something like monotone continuous functions between domains of algebraic directed-complete partial orders.

The fact that we cannot have a set-theoretic model for HOL if it has a solution to

```
"lambda  = Var num | App lambda lambda | App (lambda -> lambda)"
```

tells us that the current principles of type and term definition will not allow us to create a type and terms that satisfy this description. It does not tell us that there are no principles of definition that could be added to HOL that would be consistent and would admit a solution to this description, at least not directly. However, no such such principles can exist because any solution to the above description is inconsistent with HOL. The next two sections are devoted to proving a generalization of this fact and discussing some of the consequences for attempts to extend the `define_type` package.

3 The HOL Proof that the Function Space Can't Be Embedded

The precise result we show is that the following is a theorem of HOL:

$$\vdash (\exists \mathsf{x}{:}\beta\ \mathsf{y}{:}\beta.\ \neg(\mathsf{x} = \mathsf{y})) \Longrightarrow \forall \mathsf{f}{:}(\alpha \to \beta) \to \alpha.\ \neg(\mathsf{ONE\_ONE}\ \mathsf{f})$$

The reason this shows the non-existence of a solution of the description of the type `(==':lambda'==)` is because `(==':lambda'==)` must have at least two distinct elements since it has two distinct operators, and hence the function `(--'App:(lambda -> lambda) -> lambda'--)` cannot be one-to-one.

In order to prove

$$\vdash (\exists \mathsf{x}\ \mathsf{y}{:}\beta.\ \neg(\mathsf{x} = \mathsf{y})) \Longrightarrow \forall \mathsf{f}{:}(\alpha \to \beta) \to \alpha.\ \neg(\mathsf{ONE\_ONE}\ \mathsf{f})$$

we factor it through a specific instance, namely:

$$\vdash \forall \mathsf{f}{:}(\alpha \to \mathsf{bool}) \to \alpha.\ \neg(\mathsf{ONE\_ONE}\ \mathsf{f})$$

To see that we can acquire the general result, suppose that β is a type with at least two elements $\mathsf{x}{:}\beta$ and $\mathsf{y}{:}\beta$, and fix $\mathsf{f}{:}(\alpha \to \beta) \to \alpha$. Then we know by the special case

$$\vdash \neg(\mathsf{ONE\_ONE}\ \lambda \mathsf{g}.\ \mathsf{f}(\lambda \mathsf{z}.\ \mathsf{g}\,\mathsf{z} \Rightarrow (\mathsf{x}{:}\beta) \mid \mathsf{y})).$$

Therefore, there exist functions $\mathsf{g}{:}\alpha \to \mathsf{bool}$ and $\mathsf{h}{:}\alpha \to \mathsf{bool}$ such that

$$\vdash (\neg(\mathsf{g} = \mathsf{h})) \wedge (\mathsf{f}(\lambda \mathsf{z}.\ \mathsf{g}\,\mathsf{z} \Rightarrow \mathsf{x} \mid \mathsf{y}) = \mathsf{f}(\lambda \mathsf{z}.\ \mathsf{h}\,\mathsf{z} \Rightarrow \mathsf{x} \mid \mathsf{y})).$$

To show that $\neg(\mathsf{ONE\_ONE\ f})$, we therefore need to show

$$\begin{bmatrix} \neg(\mathsf{x} = \mathsf{y}) \\ \mathsf{f}(\lambda \mathsf{z}.\ \mathsf{g}\ \mathsf{z} \Rightarrow \mathsf{x} \mid \mathsf{y}) = \mathsf{f}(\lambda \mathsf{z}.\ \mathsf{h}\ \mathsf{z} \Rightarrow \mathsf{x} \mid \mathsf{y}) \\ \neg(\mathsf{g} = \mathsf{h}) \end{bmatrix} \vdash \exists \mathsf{x}_1\ \mathsf{x}_2.\neg((\mathsf{f}\ \mathsf{x}_1 = \mathsf{f}\ \mathsf{x}_2) \Longrightarrow (\mathsf{x}_1 = \mathsf{x}_2))$$

Let x_1 be $\lambda \mathsf{z}.\ \mathsf{g}\ \mathsf{z} \Rightarrow \mathsf{x} \mid \mathsf{y}$ and let x_2 be $\lambda \mathsf{z}.\ \mathsf{h}\ \mathsf{z} \Rightarrow \mathsf{x} \mid \mathsf{y}$. Then we must show that

$$[\neg(\mathsf{x} = \mathsf{y}),\ \neg(\mathsf{g} = \mathsf{h})] \vdash \neg((\lambda \mathsf{z}.\ \mathsf{g}\ \mathsf{z} \Rightarrow \mathsf{x} \mid \mathsf{y}) = (\lambda \mathsf{z}.\ \mathsf{h}\ \mathsf{z} \Rightarrow \mathsf{x} \mid \mathsf{y}))$$

To show that these functions are not equal, it suffices to show there is some term $\mathsf{w} : \alpha$ on which they differ. Since g and h differ, there exists some term on which they differ. This is the term we need. Thus we need to show

$$[\neg(\mathsf{x} = \mathsf{y}),\ \neg(\mathsf{gw} = \mathsf{hw})] \vdash \neg((\mathsf{g}\ \mathsf{w} \Rightarrow \mathsf{x} \mid \mathsf{y}) = (\mathsf{h}\ \mathsf{w} \Rightarrow \mathsf{x} \mid \mathsf{y}))$$

This follows by a simple case analysis. Therefore,

$$\vdash \neg(\mathsf{ONE\_ONE}\ (\mathsf{f} : (\alpha \to \beta) \to \alpha))$$

Hence, we are left showing the specific case

$$\vdash \forall \mathsf{f} : (\alpha \to \mathsf{bool}) \to \alpha.\neg(\mathsf{ONE\_ONE\ f})$$

That is, we must show that the powerset of a type cannot be injected into that type.

Fix a function $\mathsf{f} : (\alpha \to \mathsf{bool}) \to \alpha$. To show

$$\vdash \neg(\mathsf{ONE\_ONE\ f})$$

is equivalent to showing

$$\vdash \exists \mathsf{s}_1\ \mathsf{s}_2 : \alpha \to \mathsf{bool}.\ (\mathsf{f}\ \mathsf{s}_1 = \mathsf{f}\ \mathsf{s}_2) \wedge \neg(\mathsf{s}_1 = \mathsf{s}_2))$$

Let s_1 be $\lambda \mathsf{z} : \alpha.\ \exists \mathsf{s}.\neg((\mathsf{z} = \mathsf{f}\ \mathsf{s}) \Longrightarrow \mathsf{s}\ \mathsf{z})$. Before determining what s_2 should be, let us show that $\mathsf{s}_1(\mathsf{f}\ \mathsf{s}_1)$, *i.e.*

$$\vdash \exists \mathsf{s} : \alpha \to \mathsf{bool}.\ \neg((\mathsf{f}\ \mathsf{s}_1 = \mathsf{f}\ \mathsf{s}) \Longrightarrow \mathsf{s}(\mathsf{f}\ \mathsf{s}_1))$$

which is the same as showing

$$\vdash \neg\forall \mathsf{s} : \alpha \to \mathsf{bool}.\ (\mathsf{f}\ \mathsf{s}_1 = \mathsf{f}\ \mathsf{s}) \Longrightarrow \mathsf{s}(\mathsf{f}\ \mathsf{s}_1)$$

To see this, let us suppose to the contrary and derive a contradiction. Then we are assuming $\forall \mathsf{s} : \alpha \to \mathsf{bool}.\ (\mathsf{f}\ \mathsf{s}_1 = \mathsf{f}\ \mathsf{s}) \Longrightarrow \mathsf{s}(\mathsf{f}\ \mathsf{s}_1)$ (which is the same thing as $\neg(\mathsf{s}_1(\mathsf{f}\ \mathsf{s}_1))$). Therefore, we have it for the particular case of $\mathsf{s}_1 = \lambda \mathsf{z} : \alpha.\exists \mathsf{s}.\ \neg((\mathsf{z} = \mathsf{fs}) \Longrightarrow \mathsf{sz})$. This specific case, after beta-reduction, simplifies to $\neg\forall \mathsf{s} : \alpha \to \mathsf{bool}.\ (\mathsf{f}\ \mathsf{s}_1 = \mathsf{f}\ \mathsf{s}) \Longrightarrow \mathsf{s}(\mathsf{f}\ \mathsf{s}_1)$ (that is, $\mathsf{s}_1(\mathsf{f}\ \mathsf{s}_1)$), which contradicts our original assumption. (This argument is the part that I am referring to as a modification of Russell's Paradox.) With this contradiction we have established that

$$\vdash \exists \mathsf{s} : \alpha \to \mathsf{bool}.\ \neg((\mathsf{f}\ \mathsf{s}_1 = \mathsf{f}\ \mathsf{s}) \Longrightarrow \mathsf{s}(\mathsf{f}\ \mathsf{s}_1))$$

Let s_2 be such an s. Then we need to show

$$[\mathsf{s}_1(\mathsf{f}\ \mathsf{s}_1),\ \mathsf{f}\ \mathsf{s}_1 = \mathsf{f}\ \mathsf{s}_2,\ \neg(\mathsf{s}_2(\mathsf{f}\ \mathsf{s}_1))] \vdash (\mathsf{f}\ \mathsf{s}_1 = \mathsf{f}\ \mathsf{s}_2) \wedge \neg(\mathsf{s}_1 = \mathsf{s}_2)$$

Simplifying, we need to show

$$[\mathsf{s}_1(\mathsf{f}\ \mathsf{s}_1),\ \mathsf{f}\ \mathsf{s}_1 = \mathsf{f}\ \mathsf{s}_2,\ \neg(\mathsf{s}_2(\mathsf{f}\ \mathsf{s}_1))] \vdash \neg(\mathsf{s}_1 = \mathsf{s}_2)$$

However, this follows immediately since we have $\mathsf{s}_1(\mathsf{f}\ \mathsf{s}_1)$ and $\neg(\mathsf{s}_2(\mathsf{f}\ \mathsf{s}_1))$. Therefore $\mathsf{f} : \alpha \to \mathsf{bool}$ is not one-to-one.

4 Constraints on type constructors that don't work

In the previous section, we showed that a certain class of functions in HOL cannot be one-to-one. This puts limitations on possible extensions to the class of specifications to which `define_type` might be extended. Specifications of the form

$$\mathsf{ty} = \ldots \mid \mathsf{C_i} \ldots (\mathsf{ty} \to \tau) \ldots \mid \ldots$$

are obviously ruled out by the result in the previous section, since $\mathsf{C_i}$ could not be a one-to-one function (assuming τ has at least two elements). On the other hand, there is nothing to rule out specifications of the form

$$\mathsf{ty} = \ldots \mid \mathsf{C_i}\ \ldots (\tau \to \mathsf{ty}) \ldots \mid \ldots$$

provided that ty does not occur in τ, and indeed it is possible to extend `define_type` to allow such cases. It is tempting to generalize these results to the conclusion that positive occurrences of the type being defined in the argument types of the constructors for the type being defined are allowable. However, the result from the previous section can be used to see that this is not the case. Consider a specification of the form

$$\mathsf{ty} = \ldots \mid \mathsf{C_i}\ ((\mathsf{ty} \to \tau_1) \to \tau_2) \mid \ldots$$

where both τ_1 and τ_2 are types not involving ty and each having at least two elements. Then ty has only a positive occurrence. However, if there were a solution to this specification, the constructor $\mathsf{C_i}$ would have to be a one-to-one function. Since τ_2 has at least two elements, there exists a one-to-one function, $\mathsf{f} : (\mathsf{ty} \to \tau_1) \to ((\mathsf{ty} \to \tau_1) \to \tau_2)$. Thus, if $\mathsf{C_i}$ were a one-to-one function, the composition $(\mathsf{C_i} \circ \mathsf{f}) : (\mathsf{ty} \to \tau_1) \to \tau_1$ would be a one-to-one function, contradicting the result of the previous section, since τ_1 is assumed to have at least two elements.

There exists another problem with trying to use a condition such as saying that a specification is acceptable provided that there are no negative occurrences of the type being defined among the argument types of the constructors. A "negative occurrence" is only referring to occurrences within the function space type constructor. What conditions are to be placed on occurrences of the type being defined within other type constructors? One possible attempt is to say that the recursive occurrences of the type being defined cannot be within a type constructor if the polymorphic variable for which the type being defined is being substituted occurred under a function space arrow in the definition of the type constructor. This is either way too much or not sufficient, depending upon how you interpret the statement. If only explicit occurrences of type variables under function space arrows are counted when examining the definitions of type constructors, this is not sufficient. Using the principle of type definition in HOL, we can create new type constructors that are as problematic as the function space arrow. (For example, we can create a type isomorphic to, but distinct from, the function space.) When this troublesome type constructor is used in defining another type constructor, we cannot allow recursive occurrences of the type being defined within it, even though it is not the function space type constructor itself. Thus, the restriction is not sufficient to guarantee that we will be able to solve specifications satisfying it.

Alternately, we could recursively calculate whether a type variable occurs "implicitly" within the function space arrow in the definition of a type constructor. However, if pursued uniformly, this approach is destined to failure. All non-vacuous polymorphism for type constructors currently in the core HOL system is derived from the polymorphism of the function space arrow. If we attempt to rule out all type constructors whose type variables have implicit occurrences within function space arrows, we will end up ruling out all type constructors currently in main use, leading us to no real extension whatsoever. For example, $(\alpha, \beta)\mathsf{prod}$ is defined in terms of a predicate on $\alpha \to \beta \to \mathsf{bool}$ and $(\alpha, \beta)\mathsf{sum}$ is defined in terms a predicate on the type $\mathsf{bool} \to \alpha \to \beta \to \mathsf{bool}$, so even the product and sum constructors would be ruled out. To a certain extent, the function space constructor is a red herring. It is the source of all difficulties, but then again it is the principle source of polymorphism. Clearly, some other solution needs to be sought for the problem of within which type constructors to allow recursive occurrences of a type being define.

5 What we can build

To get a handle on how to extend `define_type` to allow recursive occurrences of the type being define within some type constructors, let us recall another possible extension, namely to mutually recursive types. With mutually recursive type definitions, the result we need to use the mutually recursive types is the initiality theorem, which allows us to define functions over the mutually recursive types by mutual recursion and by cases. Tom Melham gives the following example in an email message sent to `info-hol`, 26 April 1992. Consider the following mutually recursive type specification:

$$\mathsf{a} = \mathsf{A}_1 \mid \mathsf{A}_2\ \mathsf{b} \quad \text{and} \quad \mathsf{b} = \mathsf{B}_1 \mid \mathsf{B}_2\ \mathsf{a}$$

For this type specification, the initiality theorem we need is

$$\begin{array}{l}\vdash \forall \mathsf{x}_1 : \alpha\ \ \mathsf{x}_2 : \beta\ \ \mathsf{f}_1 : \beta \to \mathsf{b} \to \alpha\ \ \mathsf{f}_2 : \alpha \to \mathsf{a} \to \beta. \\ \quad \exists!(\mathsf{fn}_1, \mathsf{fn}_2).(\mathsf{fn}_1\ \mathsf{A}_1 = \mathsf{x}_1) \wedge \\ \qquad (\forall \mathsf{b}.\ \mathsf{fn}_1\ (\mathsf{A}_2\ \mathsf{b}) = \mathsf{f}_1\ (\mathsf{fn}_2\ \mathsf{b})\ \mathsf{b}) \wedge \\ \qquad (\mathsf{fn}_2\ \mathsf{B}_1 = \mathsf{x}_2) \wedge \\ \qquad (\forall \mathsf{a}.\ \mathsf{fn}_2\ (\mathsf{B}_2\ \mathsf{a}) = \mathsf{f}_2\ (\mathsf{fn}_1\ \mathsf{a})\ \mathsf{a})\end{array}$$

It would not be sufficient to give back only the two theorems:

$$\vdash \forall \mathsf{x}_1 : \alpha\ \ \mathsf{f}_1 : \mathsf{b} \to \alpha.\ \ \exists!\mathsf{fn}_1.(\mathsf{fn}_1\ \mathsf{A}_1 = \mathsf{x}_1) \wedge (\forall \mathsf{b}.\ \mathsf{fn}_1\ (\mathsf{A}_2\ \mathsf{b}) = \mathsf{f}_1\ \mathsf{b})$$

and

$$\vdash \forall \mathsf{x}_2 : \beta\ \ \mathsf{f}_2 : \mathsf{a} \to \beta.\ \ \exists!\mathsf{fn}_2.(\mathsf{fn}_2\ \mathsf{B}_1 = \mathsf{x}_2) \wedge (\forall \mathsf{a}.\ \mathsf{fn}_2\ (\mathsf{B}_2\ \mathsf{a}) = \mathsf{f}_2\ \mathsf{a})$$

We need to be able to make mutually recursive function definitions over our mutually recursive types.

The same situation holds for the cases of types defined with recursive occurrences within type constructors. To see this, let us consider another example:

$$\mathsf{Tree} = \mathsf{LEAF}\ \mathsf{bool} \mid \mathsf{NODE}\ (\mathsf{Tree}\ \mathsf{list})$$

and suppose we want to define the function that takes the conjunction of all the leaves of a tree. If the only theorem allowing us to define functions over trees is

$$\begin{array}{l}\vdash \forall \mathsf{f}:\mathsf{bool}\to\alpha\ \ \mathsf{g}:\mathsf{Tree\ list}\to\alpha.\\ \quad \exists!\mathsf{fn}.(\forall \mathsf{b}.\ \mathsf{fn}\ (\mathsf{LEAF\ b}) = \mathsf{f\ b})\wedge\\ \qquad (\forall \mathsf{tl}.\ \mathsf{fn}\ (\mathsf{NODE\ tl}) = \mathsf{g\ tl}),\end{array}$$

then it will not be possible for us to make our definition. We need a stronger principle, namely:

$$\begin{array}{l}\vdash \forall \mathsf{x}:\beta\ \ \mathsf{f_1}:\mathsf{bool}\to\alpha\ \ \mathsf{f_2}:\beta\to\mathsf{Tree\ list}\to\alpha\ \ \mathsf{f_3}:\alpha\to\beta\to\mathsf{Tree}\to\mathsf{Tree\ list}\to\beta.\\ \quad \exists!(\mathsf{fn_1},\mathsf{fn_2}).(\forall \mathsf{b}.\ (\mathsf{fn_1}\ (\mathsf{LEAF\ b}) = \mathsf{f_1\ b})\wedge\\ \qquad (\forall \mathsf{tl}.\ \mathsf{fn_1}\ (\mathsf{NODE\ tl}) = \mathsf{f_2}\ (\mathsf{fn_2\ tl})\ \mathsf{tl})\wedge\\ \qquad (\mathsf{fn_2\ NIL} = \mathsf{x})\wedge\\ \qquad (\forall \mathsf{t\ tl}.\ \mathsf{fn_2}\ (\mathsf{CONS\ t\ tl}) = \mathsf{f_3}\ (\mathsf{fn_1\ t})\ (\mathsf{fn_2\ tl})\ \mathsf{t\ tl})\end{array}$$

With this principle, we can make the desired function definition by

$$\begin{array}{l}(\forall \mathsf{b}.\ \mathsf{tree\_conj}(\mathsf{LEAF\ b}) = \mathsf{b})\wedge\\ (\forall \mathsf{tl}.\ \mathsf{tree\_conj}(\mathsf{NODE\ tl}) = \mathsf{map\_tree\_conj\ tl})\wedge\\ (\mathsf{map\_tree\_conj\ NIL} = \mathsf{T})\wedge\\ (\forall \mathsf{t\ tl}.\ \mathsf{map\_tree\_conj}(\mathsf{CONS\ t\ tl}) = ((\mathsf{tree\_conj\ t})\wedge(\mathsf{map\_tree\_conjtl})))\end{array}$$

In order to be able to derive such an initiality theorem, precisely what we need is the initiality theorem for lists. More generally, if we are given an initiality theorem for a recursively defined polymorphic type constructor (possibly a type constructor defined by mutual recursion with other types) then we will be able to solve further mutually recursive type specifications which make free use of this constructor. The construction in general makes use of the ability to make mutually recursive type definitions. To see how, we will sketch the construction for the specific case of the tree example.

To solve the tree specification, using the initiality theorem for lists, we transform it into the mutually recursive specification:

$$\mathsf{Tree} = \mathsf{LEAF'\ bool} \mid \mathsf{NODE'}\ (\mathsf{Tree\_list})$$

and

$$\mathsf{Tree\_list} = \mathsf{Tree\_NIL} \mid \mathsf{Tree\_CONS\ Tree\ Tree\_list}.$$

From here, we show that Tree_list is isomorphic to Tree list, and use this isomorphism to define LEAF, NODE, and to translate the initiality theorem for this mutual recursion into the initiality theorem we desire for Tree.

This last summer Healfdene Goguen and myself implemented in `hol90` an extension to `define_type` that would solve recursive specifications that involve recursive type constructors by using this general principle to translate them into mutually recursive specifications not involving the recursive type constructors. The mutually recursive specifications are then solved using a package written by Myra VanInwegen and myself following along the lines laid out by Tom Melham in his email message of 26 April 1992. This work currently only allows single recursive type specifications involving recursive type constructors. Also

it takes as input an SML datatype description rather than a string to be parsed into such a description. In future work it will be extended to allow for mutually recursive type specifications involving recursive type constructors, and to include a parser to allow concrete syntax for specifications. Because the extensions described above is fundamentally built out of the existing `define_type` package, type specifications of the form

$$\mathsf{ty} = \ldots \mid \mathsf{C_i}\ \ldots (\tau \to \mathsf{ty}) \ldots \mid \ldots$$

still cannot be handled. To allow these kinds of specifications would require changing `define_type` to be based on arbitrarily branching trees, instead of only finitely branching trees. Once such an extension is incorporated into the basic `define_type` package, it should be routine to change the mutually recursive definitions involving recursive type constructors to also include such specifications.

References

[1] M. J. C. Gordon. *The HOL System.* Cambridge Research Centre, SRI International, and DSTO Australia, 1989.

[2] T.F. Melham. 'Automating Recursive Type Definitions in Higher Order Logic', in: *Current Trends in Hardware Verification and Automated Theorem Proving*, edited by G. Birtwistle and P.A. Subrahmanyam (Springer-Verlag, 1989), pp. 341–386.

[3] T.F. Melham. Email correspondence. `info-hol` email, 26 April 1992.