Typed logics with states
van Eijck, D.J.N.

Published in:
Logic Journal of the IGPL

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Typed Logics With States

Jan van Eijck, CWI, P.O. Box 94079, 1090 GB Amsterdam, NL.
Email: jve@cwi.nl

Abstract

The paper presents a simple format for typed logics with states by adding a function for register update to standard typed lambda calculus. It is shown that universal validity of equality for this extended language is decidable (extending a well-known result of Friedman for typed lambda calculus). This system is next extended to a full fledged typed dynamic logic, and it is illustrated how the resulting format allows for very simple and intuitive representations of dynamic semantics for natural language and denotational semantics for imperative programing. The proposal is compared with some alternative approaches to formulating typed versions of dynamic logics.

Keywords: type theory, compositionality, denotational semantics, dynamic semantics

1 Introduction

A slight extension to the format of typed lambda calculus is enough to model states (assignments of values to storage cells) in a very natural way. Let a set $R$ of registers or storage cells be given. If we assume that the values to be stored all have type $T$, then the domain of the states is the set $R \rightarrow D_T$. Call this domain $D_\circ$.

If we want to be able to talk about stores in the calculus itself, we have to do two things:

1. introduce a format for talking about register assignment, and
2. make sure that we cannot assign values to types that are themselves built out of stores.

Point (1) is easy: just add expressions $(r|E)$ to the language, where $r$ is a register or store and $E$ a possible value for that store. The type of $(r|E)$ will be $(\circ,\circ)$, for the act of putting the new value $E$ in store cell $r$ effects a mapping from states to states.

To see what (2) is about, note that we our definition of $D_\circ$ becomes circular if we take $D_\circ$ to be the set $R \rightarrow D_T$, for some type $T$ with $D_T$ defined in terms of $D_\circ$. To avoid circularity we have to make sure that the types that are built using states are not among the possible values that can be stored in the memory cells.

We need not assume, of course, that all storage cells have the same type, but we must assume that the stored values are all of a type which does not depend on $D_\circ$. What we need for this is a distinction between standard types (types defined without the help of $\circ$) and extended types (types construed by means of $\circ$, possibly among other things). Storage cells or registers are expressions of type $(\circ,T)$, where $T$ is a standard type. If $r$ is a register of type $(\circ,T)$, then $[r]$, the interpretation of $r$, is in the domain $D_\circ \rightarrow D_T$, i.e., $[r]$ is a map from states to values of the stored type. Thus, under the assumption that storage cells can store items of all standard types, the domain $D_\circ$ consists of all those functions $f$ from $\bigcup_T R_{\circ T}$ to $\bigcup_T D_T$ with the property that $f(r) \in D_T$ iff $r$ has type $(\circ,T)$.
We will present the basic format of typed logic with states (TLS) in Section 2. Section 3 presents an equational calculus for TLS, extending the familiar rules for $\beta$ and $\eta$ equality with axioms for register lookup ($\sigma$ equalities), axioms for register update ($\rho$ equalities), and an axiom of register extensionality ($\tau$ equality). Section 4 proves that these axioms are sound and complete for standard models. The completeness proof uses a detour via general models along the lines of Friedman [8]. Section 5 extends the familiar $\beta\eta$ reduction from typed logic to $\beta\eta\sigma\rho\tau$ reduction for TLS. Combining the fact that $\beta\eta\sigma\rho\tau$ reduction for TLS is strongly normalizing with the completeness theorem we get the rather surprising result that the relation $\models E = F$ is decidable.

Section 6 adds equations to the language as expressions in their own right and defines the boolean operations and the universal and existential quantifiers in terms of those. The next sections show that TLS can serve as a meeting ground for programming semantics and natural language semantics. Section 7 presents a translation of while programs into TLS, and Section 8 discusses various ways of formulating a dynamic semantics for natural language fragments into TLS. Section 9 compares the proposal with some alternative approaches. The final section of the paper puts some further work on the agenda.

2 How to Extend Typed Logics with States

To define Typed Logic with States (TLS), let $B$ be a set of basic types. Then the set of standard types over $B$ is given by:

$$T ::= B | (T_1, T_2)$$

Extend the set of standard types as follows (it is assumed that $\diamond \notin B$):

$$U ::= \diamond | T | (U_1, U_2)$$

Call the members of $U - T$ extended types. The extended types are the types in which $\diamond$ occurs. We will often abbreviate a type $(U, U')$ as $UU'$.

For simplicity, we will not employ constants in the language (constants will be added in the extension of the language defined in Section 6). For every standard type $T$, registers of type $(\diamond, T)$ are allowed. As explained before, a register of type $(\diamond, T)$ is a store for items of type $T$. Let $R_{\diamond T}$ be the set of registers of type $(\diamond, T)$. Let $R$ be the set $\bigcup_T R_{\diamond T}$. We assume that there is an enumeration $r_1, r_2, \ldots$ of the members of $R$.

We allow variables in all types $U$, indeed we assume a countably infinite supply $V_U$ of them for every type $U$.

The set of expressions of our language of typed logics with states (TLS) is given by the rule (we use $v$ for the variables and $r$ for the registers):

$$E ::= v | r | (E_1E_2) | (\lambda v. E) | (r|E).$$

For every register $r$ there is a standard type $T$ such that $r$ has type $(\diamond, T)$. The formation of applications $(E_1E_2)$ is constrained by the requirement that $E_1, E_2$ must have types $(U_1, U_2)$ and $U_1$, respectively. If this requirement is met, $(E_1E_2)$ is well-formed, and its type is $U_2$. If $v$ has type $U_1$ and $E$ type $U_2$, then the type of $(\lambda v. E)$ is $(U_1, U_2)$. The type of $(r|E)$ is $(\diamond, \diamond)$. An expression $(r|E)$ is called a state changer,
because it is to be interpreted as a function from states to states which changes the
input state by assigning a new value $E$ to register $r$. The formation of state changers
$(r|E)$ is constrained by the requirement that if $r$ has type $(\circ, T)$ then $E$ must have
type $T$. Call this language $L_\circ$.

If the types of $v$, $E$ and $F$ are known, the types of $(\lambda v. E)$ and $(EF)$ are uniquely
determined, and the type of $(r|E)$ is always $(\circ, \circ)$, so we will not always write all
subscripts for types. We will also occasionally omit outer parentheses, and write
$(\lambda v. E)$ as $\lambda v. E$, and $(EF)$ as $EF$. An expression $\lambda v_1.(\lambda v_2.\ldots(\lambda v_n.E)\ldots)$ will be
written as $\lambda v_1v_2\ldots v_n.E$, and $(\ldots((EF_1)F_2)\ldots F_n)$ as $EF_1\ldots F_n$, i.e., we assume
that application, indicated by parentheses ( and ), associates to the left. Also, we
will let type subscripts do double duty as indices, by writing, e.g., $(r|Ez)E_\circ$ instead
of $(r|Ez)E_\circ'$). It is a feature of the language that expressions of different types are
different, so this habit is harmless.

3 An Equational Calculus for TLS

The equational calculus $L_\circ$ has as its formulas statements of the form $E = F$, with
$E, F \in L_\circ$. The axioms and rules employ the notion of substitution of $F$ for free
occurrences of $v$ in $E$, with notation $E[v := F]$; the formal definition is completely
standard. Also, $FV(E)$ is used for the set of variables with free occurrences in $E$;
again, the definition is routine.

Reflexivity, symmetry and transitivity of equality:

\[
\begin{array}{ccc}
E = E & E = F & F = E \\
F = E & E = F & F = G \\
E = F & F = G & G = F
\end{array}
\]

Context rules:

\[
\begin{array}{c}
E = F \\
\frac{E = F}{EG = FG} \\
\frac{E = F}{GE = GF}
\end{array}
\]

$\beta$ and $\eta$ axioms:

\[
(\lambda v. E)F = E[v := F] \\
\frac{\lambda v. Ev = E}{v \notin FV(E)}
\]

$\sigma$ axioms (for register lookup):

\[
r_i((r_i|E)F) = E \\
\frac{r_i((r_j|E)F) = r_iF}{i \neq j}
\]

$\rho$ axioms (for register update):

\[
(r_i|E)((r_j|F)G) = (r_i|E)G \\
\frac{(r_i|E)((r_j|F)G) = (r_j|F)((r_i|E)G)}{i \neq j}
\]

$\tau$ axiom (for register extensionality):

\[
(r|(rE))E = E.
\]

Only the $\sigma$, $\rho$ and $\tau$ axioms are new.

If $E = F$ can be derived with the rules from the axioms in a finite number of steps,
we say that $E = F$ is a theorem of the calculus; we indicate this with $\vdash E = F$. 

\[
\]
4 Soundness and Completeness for TLS

First we define full models for TLS. Let non-empty domains $D_b$ be given for all $b \in B$. Then the full model over $\{D_b\}$ is constructed as follows:

\[
D_{\langle T_1, T_2 \rangle} := D_{T_1} \rightarrow D_{T_2}, \\
D_\circ := \{ s \in R \rightarrow \bigcup_T D_T \mid \forall r_{\circ T} \in R \ s(r_{\circ T}) \in D_T \}, \\
D_{\langle U_1, U_2 \rangle} := D_{U_1} \rightarrow D_{U_2}.
\]

Note that in case all members of $R$ are of the same type $T$, the domain $D_\circ$ has the form $R \rightarrow D_T$. If $s \in D_\circ$, $r \in R_{\circ T}$, and $d \in D_T$, we use $s[r \mapsto d]$ for the function $f \in D_\circ$ which is given by $f(r') = d$ if $r \approx r'$ and $f(r') = s(r')$ otherwise. Note that $s[r \mapsto d] \in D_\circ$.

We refer to the full model over $\{D_b\}$ as $M = \{D_b\}$. An assignment in a full model $M = \{D_b\}$ is a function $g : V \rightarrow \bigcup_U D_U$ satisfying $g(v) \in D_{U}$ if $v \in V_U$.

Let $g$ be an assignment for $M = \{D_b\}$. Then the interpretation function $\cdot M$ in $M$ is defined as follows (note the use of $\lambda$ for ‘lambda abstraction in the metalanguage’ in what follows):

\[
\begin{align*}
[v_U]_g^M := g(v_U) \\
[r_T]_g^M := \text{the function given by } \lambda s.s(r) \\
\lambda v. E \cdot [r_T|E_T]_g^M := \text{the function given by } \lambda s.s[r_T \mapsto [E_T]_g^M] \\
(E (U_1, U_2) E_U) \cdot [r_T]_g^M := [E (U_1, U_2)]_g^M ([E_U]_g^M) \\
(\lambda v_1 E_U) \cdot [r_T]_g^M := \text{the function given by } \lambda d.[E_U]_g^M \cdot [v_1 \mapsto d].
\end{align*}
\]

Note in particular that the interpretation of a register $r_{\circ T}$ is indeed a function in $D_\circ \rightarrow D_T$, and that the interpretation of a state changer $(r|E)$ is indeed a function in $D_\circ \rightarrow D_\circ$.

If $M$ is a full TLS model, we use $M \models E = F$ for: for every assignment $g$, $[E]_g^M = [F]_g^M$, and we use $|E| = F$ for: for all full models $M$ it holds that $M \models E = F$.

Proposition 4.1 (Soundness) $L_\circ$ is sound for full TLS models: if $\vdash E = F$ then $|E| = F$.

To establish completeness, we make a detour via general TLS models. A general TLS structure is a set of non-empty domains $D_U$ (for every type $U$), a function $A_D : D_\circ \times R \rightarrow \bigcup_T D_T$, and a set of application functions $A_{U_1, U_2} : D_{U_1} \times D_{U_2} \rightarrow D_{U_2}$, satisfying the following extensionality requirements:

1. if $A_D(s, r) = A_D(s', r)$ for all $r \in R$, then $s = s'$,
2. if $a, b \in D_{U_1} \times D_{U_2}$ and for every $c \in D_{U_1}$ it holds that $A_{U_1, U_2}(a, c) = A_{U_1, U_2}(b, c)$, then $a = b$.

Note that a full TLS model is a general TLS structure where

\[
D_\circ = \{ s \in R \rightarrow \bigcup_T D_T \mid \forall r_{\circ T} \in R \ s(r_{\circ T}) \in D_T \},
\]

with $A_D(s, r)$ given by $s(r)$, and where each $D_{U_1, U_2}$ is the full function space $D_{U_1} \rightarrow D_{U_2}$, with $A_{U_1, U_2}(a, b)$ given by $a(b)$. 
Lemma 4.2 Suppose $E \in L_o$. Let

\[
|E| := \{F \mid \vdash E = F\},
\]

\[
D_U := \{\langle E \rangle \mid E \text{ has type } U\},
\]

\[
A_o([E], r) := [rE] \quad (E \text{ of type } \circ, r \in R),
\]

\[
A_U, U_2([E], [F]) := [EF] \quad (E \text{ of type } (U_1, U_2), F \text{ of type } U_1).
\]

Then $M_0 = (\{D_U\}, A_o, \{A_U, U_2\})$ is a general TLS structure.

Proof. To see that $M_0$ is well-defined, note that if $\vdash E = F$ then $E$ and $F$ have the same type. Also, if $\vdash E = E'$ and $\vdash F = F'$, then $\vdash (EF) = (E'F')$.

To see that $M_0$ satisfies the requirements for a general structure, we must check the requirements on the $A$ functions.

1. Assume $|E|, |F| \in D_U, |E| \neq |F|$. Then: $\vdash E = (r_{e_1}|G_{e_1}) \cdots (r_{e_n}|G_{e_n})E'$, where $n \geq 0$ and $E'$ is such that $|E'| \neq [(r|H)E''|]$, for any $r, H, E''$. To find such a form, just apply the $\beta$ and $\eta$ rules to simplify $E$, and when hitting on a term of the form $(r|H)G$, add $(r|H)$ to the state switcher list and go on with $G$. This process terminates by the fact that $\beta\eta$ reduction is strongly normalizing, plus the fact that $E$ mentions only finitely many registers. Similarly, we find $F = (r_{f_1}|H_{f_1}) \cdots (r_{f_m}|H_{f_m})F'$, with the same constraints. By the $\tau$ axiom we may also assume that no $G_{e_i}$ has the form $r_{e_i}E'$, and similarly, no $H_{f_j}$ has the form $r_{f_j}F'$. By the $\rho$ axioms we may assume that $(e_1, \ldots, e_n)$ and $(f_1, \ldots, f_m)$ are in increasing order and without repetitions.

If $|E'| \neq |F'|$ we are done, for then we can take any

\[
r \notin \{r_{e_1}, \ldots, r_{e_n}, r_{f_1}, \ldots, r_{f_m}\},
\]

and by the second $\sigma$ axiom we have $[rE] = [rE'] \neq [rF'] = [rF]$, where the inequality holds because of the constraint on $E', F'$.

Now assume that $|E'| = |F'|$ and that $(e_1, \ldots, e_n) = (f_1, \ldots, f_m)$. Then, by the fact that $|E| \neq |F|$ there must be a pair $(r_{e_i}|G_{e_i}), (r_{f_j}|H_{f_j})$ with $e_i = f_j$ and $[G_{e_i}] \neq [H_{f_j}]$. In this case we are done, for then by the first $\sigma$ axiom, $[r_{e_i}E] = [G_{e_i}] \neq [H_{f_j}] = [r_{e_i}F]$.

Finally, assume that $|E'| = |F'|$ and that $(e_1, \ldots, e_n) \neq (f_1, \ldots, f_m)$. Then either there is an $e_i \notin (f_1, \ldots, f_m)$ or there is an $f_j \notin (e_1, \ldots, e_n)$. Without loss of generality, assume the former. Then by the $\sigma$ axioms $[r_{e_i}E] = [G_{e_i}] \neq [r_{e_i}F'] = [r_{e_i}F]$, and we are done.

2. Assume $|E|, |E'| \in D_{U_1, U_2}$. Suppose that for every $|F| \in D_{U_1}$, we have $|[EF]| = |E'F'|$. Let $v \in V_{U_1}$ with $v \notin FV(E_1) \cup FV(E_2)$. Then $|[Ev]| = |E'v|$, so $\vdash (Ev) = (E'v)$. Therefore, $\vdash \lambda v.Ev = \lambda v.E'v$, and by means of the $\eta$ axiom and two applications of the transitivity rule we derive from this that $\vdash E = E'$, and therefore $|E| = |E'|$.

An assignment in a general structure $(\{D_U\}, A_o, \{A_U, U_2\})$ is a function $g : V \rightarrow \bigcup_U D_U$ satisfying $g(v) \in D_U$ if $v \in V_U$.

A general model is a general structure $M = (\{D_U\}, A_o, \{A_U, U_2\})$ together with an interpretation function $\llbracket \cdot \rrbracket^M$ defined on the terms of the language, such that $\llbracket E_U \rrbracket^M$ is a function from assignments to $D_U$ satisfying the following constraints (we will write $\llbracket E_U \rrbracket^M_g$ as $E_U^M_g$):

1. $E_U^M_g = g(v_U)$,
2. for all $s \in D_o$, all standard types $T$, all $r \in R_{cT}$ it holds that

$$A_oT([r_{cT}]_{g}^{M}, s) = A_o(s, r),$$

3. for all $s \in D_o$, all $r, r' \in R$ it holds that

$$A_o(A_o([\{(r|E)\}]_{g}^{M}, s), r') = \begin{cases} [E]_{g}^{M} & \text{if } r = r', \\ A_o(s, r') & \text{otherwise,} \end{cases}$$

4. $[(E_{U_1U_2})E_{U_1}]_{g}^{M} = A_{U_1U_2}([E_{U_1U_2}]_{g}^{M}, [E_{U_1}]_{g}),$

5. for all $d \in D_{U_1}, A_{U_1U_2}([\{(\lambda U_1 E_{U_2})\}]_{g}^{M}, d) = [E_{U_2}]_{g[v_{U_1} \mapsto d]}^{M}$.

It follows from the requirements on general structures that there can be at most one interpretation function for any general structure $M$ and any assignment $g$ for $M$. This is so because the requirements on general structures force the values of $\{[r_{T}]_{g}^{M}, \} of [[(r|E)]_{g}^{M} and of \{(\lambda U_1 E_{U_2})\}_{g}^{M}$ to be unique. If an interpretation for a general structure $M = \{(D_U), A_o, \{A_{U_1U_2}\}\}$ exists, then we call $M$ a general model.

We use $M \models E = F$ just in case every assignment $g$ for general model $M$ satisfies $[E]_{g}^{M} = [F]_{g}^{M}$.

A substitution $\theta$ is a mapping from variables to terms satisfying the constraint that $\theta(v_U)$ is of type $U$. This is extended to terms in the standard manner. We will use $\theta E$ for the result of applying $\theta$ to $E$. Note that $E$ and $\theta E$ have the same type. Notation for the substitution $\theta'$ that differs only from $\theta$ in the fact that $v$ is mapped to $E$ is $\theta[v \mapsto E]$. If $g$ is an assignment in the term general structure (i.e., the values $g(v)$ are term equivalence classes), then substitution $\theta$ represents $g$ if it holds for all variables $v$ that $\theta(v)$ is a representative of the equivalence class $g(v)$.

**Lemma 4.3** Let $\theta$ be a substitution that represents $g$ in the term structure $M$. Then putting $[E]_{g}^{M} = [\theta E]$ makes $M$ a general model.

**Proof.** To see that $[E]_{g}^{M} = [\theta E]$ is well-defined, observe that $\theta E_U$ has type $U$, so $[\theta E_U]_{g} \in D_U$. Next, we must check the requirements on the interpretation function.

1. $[v_{U}]_{g}^{M} = [\theta(v_{U})]$ (by the fact that $\theta$ represents $g$).

2. $[r_{T}]_{g}^{M} = [\theta(r_{T})] = [r_{T}]$. By the fact that $A_o([E], r) = [rE] = A_oT([r], [E])$ this is indeed the required element of $D_oT$.

3. The following reasoning shows that the requirement is met:

$$A_o(A_o([\{(r|E)\}]_{g}^{M}, [E_{o}]), r') = A_o(A_o([\theta(r_{T}|E_{T})], [E_{o}]), r') = A_o(A_o([\{(r|E_{T})\}], [E_{o}]), r') = A_o(\{(r|\theta E_{T})E_{o}\}, r') = [r'(\{(r|\theta E_{T})E_{o}\})] = [\theta E_{T}] \text{ if } r = r', [r'E_{o}] \text{ otherwise.}$$

4. Here is the reasoning for this case:

$$[(E_{U_1U_2})E_{U_1}]_{g}^{M} = [\theta(E_{U_1U_2})E_{U_1}] = [\theta(E_{U_1U_2})E_{U_1}] = A_{U_1U_2}([E_{U_1U_2}]_{g}^{M}, [E_{U_1}]_{g}^{M}).$$
5. Let \([E_{U_1}] \in D_{U_1}\). Then:

\[
\begin{align*}
A_{U_1U_2}([\lambda v_{U_1}.E_{U_2}])^M & = A_{U_1U_2}([\theta(\lambda v_{U_1}.E_{U_2}),[E_{U_1}])] \\
& = A_{U_1U_2}([\lambda v_{U_1}.\theta E_{U_2}], [E_{U_1}]) \\
& = [\theta v_{U_1} \mapsto E_{U_1}](E_{U_2}) \\
& = [E_{U_2}]^M_{[\theta v_{U_1} \mapsto E_{U_1}]};
\end{align*}
\]

where the last step is licensed because \(\theta v_{U_1} \mapsto E_{U_1}\) represents \(g[v_{U_1} \mapsto [E_{U_1}]]\).

This completes the check of the requirements and the proof. 

\section*{Theorem 4.4 (Generalized Completeness)}

If for every general model \(M\), \(M \models E = F\), then \(\vdash E = F\).

\textbf{Proof.} By means of the construction of a canonical general model \(M_0\). Let \(g\) be the identity assignment \(v \mapsto [v]\). Then the identity substitution \(\theta : v \mapsto v\) represents \(g\), and we have:

\[
[E] = [\theta E] = [E]^M_{M_0}.
\]

By construction we have: if \(\not\models E = F\), then \(M_0 \not\models [E] = [F]\). \hfill \Box

Let \(M = (\{D_U\}, A_0, \{A_{U_1U_2}\})\) and \(N = (\{E_U\}, B_0, \{B_{U_1U_2}\})\) be general TLS models. A system \(\{f_U\}\) is a partial homomorphism of \(M\) onto \(N\) if the following hold:

1. Each \(f_U\) is a partial surjective map from \(D_U\) onto \(E_U\).
2. If \(f_0\) is defined for \(s \in D_0\), then \(f_0(s)\) is the unique element of \(E_0\) with

\[
f_T(A_0(s,r)) = B_0(f_0(s),r),
\]

for all \(T\), all \(r \in R_{0T}\).
3. If \(f_{U_1U_2}\) is defined for \(d\) then \(f_{U_1U_2}(d)\) is the unique element of \(E_{U_1U_2}\) such that

\[
f_{U_1U_2}(A_{U_1U_2}(d,x)) = B_{U_1U_2}(f_{U_1U_2}(d),f_{U_1}(x)),
\]

for all \(x \in \text{dom } (f_{U_1})\).
4. For all \(T\), all \(r \in R_{0T}\), \(f_{0T}(\lambda s.A_0(s,r))\) is defined.
5. If \(f_0\) is defined for \(s \in D_0\) and \(f_T\) is defined for \(d \in D_T\), and \(r \in R\), then it holds that \(f_0\) is defined for \(s[r \mapsto d] \in D_0\), and \(f_0\) satisfies

\[
f_0(s[r \mapsto d]) = f_0(s)[r \mapsto f_T(d)].
\]

Note that a partial homomorphism \(\{f_U\}\) is fully determined by \(\{f_0 \mid b \in B\}\).

\textbf{Proposition 4.5} If \(M, N\) are general models and \(\{f_U\}\) a partial homomorphism of \(M\) onto \(N\) then \(f_{0T}(\lambda s.A_0(s,r)) = \lambda s.B_0(s,r)\).

\textbf{Proof.} Because \(\{f_U\}\) is a partial homomorphism, \(f_{0T}(\lambda s.A_0(s,r))\) is defined. By property (3) of partial homomorphisms, \(f_{0T}(\lambda s.A_0(s,r))\) is the unique element \(z\) of \(E_{0,T}\) such that

\[
\begin{align*}
f_T(A_{0T}(\lambda s.A_0(s,r),s)) & = f_T(A_0(s,r)) \\
( \text{property } (2)) & = B_0(f_0(s),r) \\
& = B_{0T}(z,f_0(s)),
\end{align*}
\]

for all \(s \in \text{dom } (f_0)\). Because \(f_0\) is onto, it follows that \(z = \lambda s.B_0(s,r)\). \hfill \Box
**Proposition 4.6** If $M, N$ are general models, $g$ is an $M$ assignment, $h$ an $N$ assignment, and $\{f_U\}$ a partial homomorphism of $M$ onto $N$ satisfying $f_U(g(v)) = h(v)$ for every $U$, every $v \in V_U$, then $f_U[E]^M_g = [E]^N_h$ for every term $E$ of type $U$.

**Proof.** Induction on the structure of $E$.
For $r \in R$, we have by Proposition 4.6:

$$f_O([r]^M_g) = f_O(\lambda s.A_o(s, r))$$
$$= \lambda s.B_o(s, r)$$
$$= [r]^N_h.$$

For expressions of the form $(EF)$ we have:

$$f_U([EF]^M_g) = f_U(A_UU([E]^M_g, [F]^M_g))$$
$$= B_UU([E]^N_h, [F]^N_h)$$
$$= [EF]^N_h.$$

To show $f_UU([\lambda v.E]^M_g) = [\lambda v.E]^N_h$, take $d \in \text{dom}(f_U)$. We must establish that $f_UU(A_UU([\lambda v.E]^M_g, d)) = B_UU([\lambda v.E]^N_h, f_U(d))$.

$$f_UU(A_UU([\lambda v.E]^M_g, d)) = f_UU([\lambda v.E]^M_{g[v\rightarrow d]})$$
$$= [E]^{N}_{h[v\rightarrow f_U(d)]}$$
$$= B_UU([\lambda v.E]^N_h, f_U(d)).$$

Finally, to show $f_{oo}([r|E]^M_g) = [r|E]^N_h$, take $s \in D_o$. We must show that

$$f_o(A_{oo}([r|E]^M_g, s)) = B_{oo}([r|E]^N_h, f_o(s)).$$

$$f_o(A_{oo}([r|E]^M_g, s)) = f_o(s| r \mapsto [E]^M_g)$$
$$= f_o(s) | r \mapsto f_o([E]^M_g)$$
$$= f_o(s) | r \mapsto [E]^N_h$$
$$= B_{oo}([r|E]^N_h, f_o(s)).$$

**Proposition 4.7** If there is a partial homomorphism from $M$ onto $N$, then $M \models E = F$ implies $N \models E = F$.

**Proof.** Use Proposition 4.6. 

**Proposition 4.8** If $N = \{E_U\} \cup \{A_o\} \cup \{A_UU\}$ is a general model and $M = \{D_U\}$ is a full model, and moreover $|E_b| \leq |D_b|$ for $b \in B$, then there is a partial homomorphism from $M$ onto $N$. 
5. Reducing TLS Expressions

Proof. Let \( \{ f_b \} \) be a set of arbitrary partial surjective maps from \( D_b \) to \( E_b \). First extend \( \{ f_b \} \) to all standard types, as follows. Suppose \( f_{T_1} \) and \( f_{T_2} \) have been defined. Define \( f_{T_1,T_2}(d) \) to be the unique element of \( E_{T_1,T_2} \) (if it exists) such that \( f_{T_2}(d(y)) = A_{T_1,T_2}(f_{T_1}(d),f_{T_1}(y)) \), for all \( y \in \text{dom}(f_{T_1}) \).

To see that \( f_{T_1,T_2} \) is surjective, take \( z \in E_{T_1,T_2} \), and let \( x \in D_{T_1,T_2} \) be such that for all \( y \in \text{dom}(f_{T_1}) \), \( x(y) \in f_{T_2}^{-1}(A_{T_1,T_2}(z,f_{T_1}(y))) \). (This uses the surjectivity of \( f_{T_2} \).) Then \( f_{T_1,T_2}(x) = z \).

Next define the map \( f_\circ \), by putting \( f_\circ(s) \) = the unique element of \( E_\circ \) (if it exists) with \( f_\circ(s)(r) = A_\circ(f_\circ(s),r) \), for all standard types \( T \), all \( r \in R_\circ(T) \). We can check that this map is surjective, as before.

Extend the map to all types \( U \), in the same manner as before, and check surjectivity as before.

All \( f_\lambda \) are surjective, and satisfy properties (2) and (3) by definition. Because \( M \) is full, property (4) boils down to: \( f_\lambda(T(\lambda s.s)(r)) \) is defined. To check this property, we have to show that there is a unique \( z \in E_\circ \) with \( f_\circ(y(r)) = A_\circ(z,f_\circ(y)) \), for all \( y \in \text{dom}(f_\circ) \). Clearly, \( z \) is given by \( \lambda s. A_\circ(s,r) \).

Finally, we check property (5). Assume \( f_\circ \) is defined for \( s \in D_\circ \), and \( f_\circ \) is defined for \( d \in D_T \). Assume \( r \in R_\circ(T) \). We check whether \( f_\circ(s[r \mapsto d]) \) is defined. By the construction of \( f_\circ \), this is the case iff there is a unique \( z \in E_\circ \) with \( f_\circ(s[r \mapsto d](r') = A_\circ(z,r') \) for all standard \( T \), all \( r' \in R_\circ(T) \). Clearly, this \( z \) is given by \( \lambda s.s[r \mapsto f_\circ(d)] \).

**Theorem 4.9 (Full Completeness)** If \( \models E = F \), then \( \vdash E = F \).

Proof. Suppose \( \not\models E = F \). Then, by the generalized completeness theorem, \( M_0 \not\models E = F \), where \( M_0 \) is the canonical general model. By proposition 4.8, there is a full model \( M \) of which \( M_0 \) is a partial homomorphic image. By proposition 4.7, \( M \not\models E = F \).

Note that for all full models \( M \) with infinite base domains, the relation \( M \models E = F \) coincides with the relation \( M_0 \models E = F \), where \( M_0 \) is our canonical term model. This is because any full model with large enough base domains has the canonical model as a partial homomorphic image. Indeed, any equality that is true in a large enough full model will be true in the canonical term model. The situation is completely analogous to the case of the ordinary typed lambda calculus (see Friedman [\textsuperscript{[8]}]).

5 Reducing TLS Expressions

If \( R \) is a relation on the set of expressions of \( L_\circ \) (a so-called notion of reduction), then \( R \) determines a relation \( \rightarrow^R \) of one-step \( R \) reduction in the following standard manner:

\[
\begin{align*}
(E,E') \in R & \quad \frac{E \rightarrow E'}{E \rightarrow^R E'} & \frac{E \rightarrow^R E'}{(FE) \rightarrow^R (FE')} & \frac{E \rightarrow^R E'}{(EF) \rightarrow^R (E'F)} \\
\frac{E \rightarrow^R E'}{\lambda v.E \rightarrow^R \lambda v.E'} & \frac{E \rightarrow^R E'}{(r|E) \rightarrow^R (r|E')} \\
\end{align*}
\]

\( R \) reduction (notation \( \rightarrow^R \)) is the reflexive transitive closure of \( \rightarrow^R \):
Recall that the notion of beta reduction is the relation between an expression of the form \((\lambda v.E)E'\) and the expression \(E[v := E']\). \(\beta\) reduction is the relation \(\longrightarrow\). Similarly, the notion of \(\eta\) reduction is the relation between \(\lambda v.Ev\) and \(E\) (provided \(v \notin FV(E)\)). We add three new notions of reduction to this list:

- The notion of \(\sigma\) reduction is the union of the relations

\[ (r_i((r_i|E)F), E) \]

and

\[ (r_i((r_j|E)F), r_iF), \text{ provided } i \neq j. \]

- The notion of \(\rho\) reduction is the union of the relations

\[ ((r_i|E)((r_i|F)G), (r_i|E)G) \]

and

\[ ((r_i|E)((r_j|F)G), (r_j|F)((r_i|E)G)), \text{ provided } j < i. \]

- The notion of \(\tau\) reduction is the relation \(((r|E))E, E)\).

From the soundness of the \(L_\diamond\) calculus it follows that \(\beta\) reduction, \(\eta\) reduction, \(\sigma\) reduction, \(\rho\) reduction and \(\tau\) reduction are all sound (after all, these reduction notions are nothing but directed versions of the corresponding equality axioms).

The standard reduction notion for typed logic is \(\beta\eta\) reduction i.e., reduction for the notion \(\beta \cup \eta\), and the standard results of typed logic that \(\beta\eta\) reduction is confluent and strongly normalizing extend readily to the setting of TLS:

**Proposition 5.1** \(\beta\eta\) reduction is confluent, i.e., \(E \beta\eta \longrightarrow F\) and \(E \beta\eta \longrightarrow F'\) together imply that there is a \(G\) with \(F \beta\eta \longrightarrow G\) and \(F' \beta\eta \longrightarrow G\).

**Proposition 5.2** \(\beta\eta\) reduction is strongly normalizing for TLS, i.e., every reduction sequence \(E \beta\eta \longrightarrow F \beta\eta \longrightarrow \cdots\) terminates.

Let \(\sigma\rho\tau\) be the relation of \(R\) reduction for \(R = \sigma \cup \rho \cup \tau\).

**Proposition 5.3** \(\sigma\rho\tau\) is weakly confluent, i.e., \(E \sigma\rho\tau \longrightarrow F\) and \(E \sigma\rho\tau \longrightarrow F'\) together imply that there is a \(G\) with \(F \sigma\rho\tau \longrightarrow G\) and \(F' \sigma\rho\tau \longrightarrow G\).

**Proof.** The claim is proved by a case analysis. We just give one case in full. The reasoning for the other cases is similar.

Assume \(E[r((r|E_T)E_o)] \sigma\rho\tau \longrightarrow E[E_T]\), and \(E[r(r|E_T)E_o] \sigma\rho\tau \longrightarrow F\), with \(F \neq E[E_T]\).

Then there are four sub-cases to consider.

1. The reduction \(E[r((r|E_T)E_o)] \sigma\rho\tau \longrightarrow F\) leaves subexpression \((r|E_T)E_o)\) unaffected, i.e., we can write \(F\) as \(E'[r(r|E_T)E_o]\). In this case both \(F\) and \(E[E_T]\) reduce in one \(\sigma\rho\tau\) step to \(E'[E_T]\).
2. The reduction $E[r((r|E_T)E_o)] \xrightarrow{\sigma \rho \tau} F$ removes subexpression $(r(r|E_T)E_o)$. In this case $E[E_T]$ reduces in one $\sigma \rho \tau$ step to $F$.

3. The reduction $E[r((r|E_T)E_o)] \xrightarrow{\sigma \rho \tau} F$ affects $E_T$. In this case, $F$ can be written as $E[r(r|E_T)E_o]$, and both $F$ and $E[E_T]$ reduce in one $\sigma \rho \tau$ step to $E[E_T]$.

4. The reduction $E[r((r|E_T)E_o)] \xrightarrow{\sigma \rho \tau} F$ affects $E_T$. In this case $F$ can be written as $E[r(r|E_T)E_o]$, and both $F$ and $E[E_T]$ reduce in one $\sigma \rho \tau$ step to $E[E_T]$.

**Proposition 5.4** $\sigma \rho \tau$ reduction is strongly normalizing, i.e., every reduction sequence $E \xrightarrow{\sigma \rho \tau} F \xrightarrow{\sigma \rho \tau} \cdots$ terminates.

**Proof.** Every $\sigma \rho \tau$ reduction step either reduces the number of symbols in an expression, or ($\rho$ steps of the second kind) is a step towards putting a sequence of registers in increasing order. □

**Proposition 5.5** $\sigma \rho \tau$ is confluent, i.e., $E \xrightarrow{\sigma \rho \tau} F$ and $E \xrightarrow{\sigma \rho \tau} F'$ together imply that there is a $G$ with $F \xrightarrow{\sigma \rho \tau} G$ and $F' \xrightarrow{\sigma \rho \tau} G$.

**Proof.** By an application of Newman’s lemma (see Klop [17]) from Propositions 5.3 and 5.4. □

Let $\beta \eta \sigma \rho \tau$ reduction be the relation $R \xrightarrow{\beta \eta}$ for $R = \beta \cup \eta \cup \sigma \cup \rho \cup \tau$. It follows immediately from the soundness of $L_0$ that $\beta \eta \sigma \rho \tau$ reduction is sound.

Here is an example $\beta \sigma \tau$ reduction:

$$
(\lambda p.((p|x)i)\lambda i.((r'|(r'|x))(ri))) \xrightarrow{\beta} ((\lambda i.((r'|(r'|x))(ri)))(r|x)i)
$$
$$
\xrightarrow{\beta} ((r'|(r'|x))(r|x)i)
$$
$$
\xrightarrow{\sigma} ((r'|(r'|x))x)
$$
$$
\xrightarrow{\tau} x.
$$

To prove that $\beta \eta \sigma \rho \tau$ reduction is confluent we again need to make a detour via weak confluence. This is because the combination of $\beta \eta$ reduction and $\sigma \rho \tau$ reduction is not orthogonal (see Klop [17]). Intuitively, orthogonality of two reduction relations $R$ and $S$ means that $R$ reduction steps never ‘spoil’ opportunities for $S$ reduction and vice versa. If two confluent reduction relations are orthogonal, then their union is again confluent. In our case, orthogonality fails due to the presence of $\tau$ reductions (an opportunity to apply $(r|(rE))E \rightarrow E'$ may in principle be spoiled by a reduction $E \rightarrow E'$).

**Proposition 5.6** $\beta \eta \sigma \rho \tau$ reduction is weakly confluent, i.e., $E \xrightarrow{\beta \eta \sigma \rho \tau} F$ and $E \xrightarrow{\beta \eta \sigma \rho \tau} F'$ together imply that there is a $G$ with $F \xrightarrow{\beta \eta \sigma \rho \tau} G$ and $F' \xrightarrow{\beta \eta \sigma \rho \tau} G$.

**Proof.** A case analysis similar to the analysis in the proof of Proposition 5.3. □

**Proposition 5.7** $\beta \eta \sigma \rho \tau$ is strongly normalizing, i.e., every reduction sequence $E \xrightarrow{\beta \eta \sigma \rho \tau} F \xrightarrow{\beta \eta \sigma \rho \tau} \cdots$ terminates.
Typed Logics With States

Proof. The proof is a straightforward adaptation of the proof of strong normalization for typed lambda calculus. See Barendregt [1], Appendix A, or Hindley and Seldin [14].

Proposition 5.8 \( \beta_\eta \mu \rightarrow \rightarrow \) is confluent, i.e., \( E \beta_\eta \mu \rightarrow \rightarrow F \) and \( E \beta_\eta \mu \rightarrow \rightarrow F' \) together imply that there is a \( G \) with \( F \beta_\eta \mu \rightarrow \rightarrow G \) and \( F' \beta_\eta \mu \rightarrow \rightarrow G \).

Proof. Again by an application of Newman’s lemma from Propositions 5.6 and 5.7.

Theorem 5.9 The relation \( \vdash E = F \) is recursive.

Proof. Immediate from the fact that \( \beta_\eta \mu \) reduction is strongly normalizing. Just reduce \( E \) and \( F \) to find out if their normal forms are the same (modulo changes in bound variables).

Theorem 5.10 The relation \( \models E = F \) is recursive.

Proof. Immediate from Theorem 5.9 and the Completeness Theorem 4.9.

It should be noted that deciding equality in TLS is not cheap, for Statman’s result that the typed lambda calculus is not elementary recursive [26] applies to TLS as well.

6 The Logical Theory of TLS

Equations in typed logic have the form \( E = F \), where \( E, F \) are assumed to be of the same type. If we assume a basic type \( t_i \) with \( D_t = \{1, 0\} \), we can consider equations as terms of type \( t \), by extending the language with a rule:

\[
E_t ::= E_U = F_U
\]

Note that this language extension boils down to adding constants \( Q(U,(U,t)) \) for every type \( U \), and writing the term \( ((Q(U,(U,t))E_U)F_U) \) as \( E_U = F_U \). The interpretation of \( E_U = F_U \) in (standard or general) model \( M \), for assignment \( g \), is given by (note the use of \textit{equals} for ‘identity at the meta-level’):

\[
\begin{cases}
1 & \text{if } [E_U]^M_g \text{ equals } [F_U]^M_g, \\
0 & \text{if } [E_U]^M_g \text{ does not equal } [F_U]^M_g.
\end{cases}
\]

We agree to write \( E_t = F_t \) as \( E \leftrightarrow F \). Call an expression of type \( t \) a formula. Some useful abbreviations for formulas and formula-forming operators can now be given:

\[
\begin{align*}
\top & ::= (\lambda x_1.x_1) \\
\bot & ::= (\lambda x_1.x_1) (\lambda x_1.\top) \\
\neg & ::= (\lambda x_1.(x_1 \leftrightarrow \bot)) \\
\land & ::= (\lambda x_1 y_1.(\lambda z_1(z_1 \leftrightarrow y_1)) = (\lambda z_1(z_1 \land z_1))) \\
\forall U E & ::= (\lambda x_U . E) (\lambda x_U . \top)
\end{align*}
\]

Write \( (E \land F) \) for \(((E \land F))\), and abbreviate \( \neg\neg E \land \neg F \) as \( (E \lor F) \), \( \neg(E \land \neg F) \) as \( (E \rightarrow F) \), and \( \neg\forall x_U \neg E \) as \( \exists x_U E \). \( \forall x_1 (\forall x_2 \ldots (\forall x_n E \ldots)) \) will be written as
∀x_1 x_2 \ldots x_n E$, and similarly for $\exists$. We will also omit brackets in $n$-ary conjunctions and disjunctions, so we write $E_1 \land \ldots \land E_n$ and $E_1 \lor \ldots \lor E_n$.

A formula $E_t$ is valid if $[E_t]_g^M = 1$ for every choice of $M, g$ ($M$ a standard model).

A logical calculus for this extended TLS language can be defined by adding the following axioms and rules to the equational calculus (see e.g. Gallin [9]).

Axiom scheme for the Booleans (giving an explicit definition of $D_t$ as $\{0, 1\}$):

$$(E_t \top \land E_t \bot) \iff \forall x_t E_t x.$$  

Extensionality expressed by means of universal quantification:

$$\forall x(Ex = Fx) \iff (E = F).$$  

Axiom schemes for fixing the meanings of the constants $Q(U, (U, t))$:

$$v_U = w_U \rightarrow E_U v_U \iff E_U w_U.$$  

Inference rule ($B'_t$ is the result of replacing an occurrence of $E_U$ in $B_t$ by $E'_U$):

$$E_U = E'_U \quad \frac{B_t}{B'_t}.$$  

This logic is not complete for full models, for the extended language is expressive enough to define the standard model of arithmetic, and the existence of a complete logic for TLS would contradict Tarski’s theorem of the non-axiomatizability of arithmetical truth.

Still, we know that the incomplete logic of types and states contains the TLS equational calculus. Also, we have completeness for general models (as can be shown by a standard extension of the completeness proof given above; see also Henkin [13]). This means that universal validities that continue to hold in all general models are provable in the logic.

As an example, we mention that for every register $r \in R_{\circ T}$ of the language the following formula is valid and provable:

$$\forall i_0 \forall x_T \exists j_0 (j = (r|x)i).$$

This is certainly valid, by virtue of the definition of the domain $D_{\circ}$. The principle is also provable, for it continues to hold in general models.

The truth of the formula illustrates that the construction of the typed domains itself guarantees that there are ‘enough states’, and that no update axiom in the style of [15] is needed, or rather, that such an update axiom is ‘provided algebraically’ by the $(r|E)$ functions and the $\sigma, \rho, \tau$ principles governing their behaviour.

It is instructive to check how the notion of equality works out for the types $(\circ, T)$. Note that the following formula is valid (and indeed an axiom of the TLS logic):

$$(E_{\circ T} = F_{\circ T}) \iff \forall x_{\circ T}(Ex = Fx).$$

This axiom says that two functions $g, h$ in $D_\circ \rightarrow D_T$ are equal iff for every $s \in D_\circ$, $g(s) = h(s)$.

**Proposition 6.1** Let $r, r' \in R_{\circ T}$. On the assumption that $|D_T| \geq 2$, the formula $r = r'$ evaluates to true in general model $M$ iff $r$ and $r'$ are the same register.
636  

Typed Logics With States

7 Denotational Semantics in TLS

The assignment statement \( r := t \) is the basic building block of imperative programming. Therefore, an analysis of imperative programs in a language that has register assignment in its algebraic basis is in a sense more natural than analyzing the assignment statement \( r := t \) in a more roundabout way. In this section we demonstrate the semantic analysis of \textit{while} programs in TLS.

Let \( z \) be the type of integers, and assume that \( r \) ranges over \( R_{oz} \). Then the following defines the language of \textit{while} programs with program variables taken from the set \( R_{oz} \).

\[
N ::= 0 | \cdots | 9 | N0 | \cdots | N9
\]

\[
A ::= r | N | (A_1 + A_2) | (A_1 - A_2) | (A_1 \diamond A_2)
\]

\[
B ::= A_1 = A_2 | A_1 < A_2 | A_1 \leq A_2 | \neg B | (B_1 \land B_2)
\]

\[
S ::= r := A | \text{skip} | (S_1; S_2) \\
\]  

| if \( B \) then \( S_1 \) else \( S_2 \) | while \( B \) do \( S \)

The meanings of the statements of the \textit{while} language are partial functions over \( R_{oz} \rightarrow \mathbb{Z} \), i.e., members of \((R_{oz} \rightarrow \mathbb{Z}) \leftarrow (R_{oz} \rightarrow \mathbb{Z})\), where \( \mathbb{Z} \) is used for the set of integers. Recall that if we build a TLS logic over basic type set \( \{z, t\} \), and assume that \( R_{oz} \) is the set of all registers, then the domain \( D_o \) has the form \( R_{oz} \rightarrow \mathbb{Z} \). Thus, the members of \((R_{oz} \rightarrow \mathbb{Z}) \leftarrow (R_{oz} \rightarrow \mathbb{Z})\) are in fact members of \( D_o \leftarrow D_o \). We can represent a member of \( D_o \leftarrow D_o \) (a partial function from states to states) by means of its graph, which is a member of \( D_{(o,(o,t))} \). Meanings of \textit{while} statements can be represented as members of \( D_{(o,(o,t))} \). We will now show how one can refer to them by means of TLS expressions in a straightforward way.

First define a relation \( \sqsubseteq \) on \( D_{(o,(o,t))} \) by means of:

\[
F \sqsubseteq G \iff \forall ij(F_{ij} \rightarrow G_{ij}),
\]
DENOTATIONAL SEMANTICS IN TLS

where $F, G$ are expressions of type $(\diamond, (\diamond, t))$, and $i, j$ variables of type $\diamond$.

Next, define a function $\text{FIX}$ in

$$(D_{(\diamond, (\diamond, t))} \rightarrow D_{(\diamond, (\diamond, t))}) \rightarrow (D_{(\diamond, (\diamond, t))} \rightarrow D_t)$$

by means of:

$$\text{FIX} := \lambda F g. \forall i j (F g i j = g i j),$$

where $F$ is a variable of type $((\diamond, (\diamond, t)), (\diamond, (\diamond, t)))$, $g$ a variable of type $(\diamond, (\diamond, t))$, and $i, j$ are variables of type $\diamond$.

If $H$ has type $D_{(\diamond, (\diamond, t))} \rightarrow D_{(\diamond, (\diamond, t))}$ and $h$ type $D_{(\diamond, (\diamond, t))}$, then $\text{FIX} H h$ has type $D_t$. The expression $\text{FIX} H h$ reduces to:

$$\forall su (H h su = h su),$$

which is true iff (the interpretation of) $h$ is the graph of a fixed point of (the interpretation of) $H$.

Finally, define a function $\mu$ in

$$(D_{(\diamond, (\diamond, t))} \rightarrow D_{(\diamond, (\diamond, t))}) \rightarrow D_{(\diamond, (\diamond, t))}$$

by means of:

$$\mu := \lambda F i j. \exists g (\text{FIX} F g \land \forall d (\text{FIX} F d \rightarrow g \sqsubseteq d) \land g i j),$$

where $F$ a variable of type $((\diamond, (\diamond, t)), (\diamond, (\diamond, t)))$, $g, d$ are variables of type $(\diamond, (\diamond, t))$, and $i, j$ variables of type $\diamond$.

If $H$ has type $D_{(\diamond, (\diamond, t))} \rightarrow D_{(\diamond, (\diamond, t))}$, then $\mu H$ has type $D_{(\diamond, (\diamond, t))}$, and $\mu H$ reduces to

$$\lambda su. \exists g (\text{FIX} H g \land \forall d (\text{FIX} H d \rightarrow g \sqsubseteq d) \land g su).$$

This expression denotes the graph of the least fixed point of (the interpretation of) $H$, if it exists, and the empty relation on $D_0$ otherwise.

The translation of $\textbf{while}$ to TLS now proceeds in four stages: first we translate numerals into type $z$, next arithmetical expressions into type $(\diamond, z)$, then boolean expressions into type $(\diamond, t)$, and finally statements into type $(\diamond, (\diamond, t))$.

Translation of numerals into type $z$ (we assume that we have a 0 for zero, that $s$ names the successor function, that $+, -, \times$ are names of the functions for addition, subtraction and multiplication in $\mathbb{Z}$, and we abbreviate $s0$ as 1, \ldots, $s\ldots sssssssss0$ as 9 and $s\ldots sssssssssss0$ as 10.

\[
\begin{align*}
0^\circ & := 0 \\
\vdots \\
9^\circ & := 9 \\
(N0)^\circ & := N^\circ \times 10 \\
\vdots \\
(N9)^\circ & := (N^\circ \times 10) + 9
\end{align*}
\]
Translation of arithmetical expressions into type $(\varnothing, z)$, using the functions $+,-,\times$ on the domain $\mathbb{Z}$:

$$
N^* := \lambda i. N^\circ \\
(\mathbf{r})^* := r \\
(A_1 + A_2)^* := \lambda i. (A_1^* i + A_2^* i) \\
(A_1 - A_2)^* := \lambda i. (A_1^* i - A_2^* i) \\
(A_1 \times A_2)^* := \lambda i. (A_1^* i \times A_2^* i)
$$

Note that the translation of $\mathbf{r}$ is $\mathbf{r}$ itself. This is correct, for $\mathbf{r}$ is a variable in the programming language, but a register in the TLS language, and it has type $(\varnothing, z)$.

Translation of boolean expressions into type $(\varnothing, t)$. We already have $=$ in type $(z, (z, t))$. Here we assume that we also have a relation $<$ in $(z, (z, t))$ available. $\leq$ is then defined as $\lambda PQ. (P < Q \lor P = Q)$.

$$
(A_1 = A_2)^* := \lambda i. (A_1^* i = A_2^* i) \\
(A_1 < A_2)^* := \lambda i. (A_1^* i < A_2^* i) \\
(A_1 \leq A_2)^* := \lambda i. (A_1^* i \leq A_2^* i) \\
(\neg B)^* := \lambda i. (\neg (B^* i)) \\
(B_1 \land B_2)^* := \lambda i. (B_1^* i \land B_2^* i)
$$

Translation of statements into type $(\varnothing, (\varnothing, t))$:

$$
(r := A)^* := \lambda ij. ((r|(A^* i))i = j) \\
\text{skip}^* := \lambda ij. (i = j) \\
(S_1; S_2)^* := \lambda ij. \exists k (S_1^* ik \land S_2^* kj) \\
(\text{if } B \text{ then } S_1 \text{ else } S_2)^* := \lambda ij. ((B^* i \land S_1^* ij) \lor (\neg B^* i \land S_2^* ij)) \\
(\text{while } B \text{ do } S)^* := (\mu \lambda gi j. ((B^* i \land \exists k (S^* ik \land gk j)) \lor (\neg B^* i \land i = j)))
$$

**Proposition 7.1** The translation $S^*$ is correct in the sense that the interpretation of $S^*$ corresponds to the semantic function specified for $S$ by the standard semantics for the while language.

**Proof.** First specify the semantics of while by some other means, e.g., in operational style, by means of transition rules, and then engage in a lengthy induction exercise (see e.g. Plotkin [24], or the textbook presentation in Nielsen and Nielsen [23]).

The proposition shows that TLS provides a cheap way of representing the meanings of while programs (by means of simpleton domains, so to speak).

The purpose of this section was to demonstrate that reflexive domains (domains $D$ that are isomorphic to the function space $[D \rightarrow D]$ of all continuous functions on $D$, proposed as a denotational semantics for programming in Scott and Strachey [25]) are by no means essential for providing a denotational semantics of recursion in imperative programming. The same point is made at greater length in Muskens [22], but with the help of a higher order logic that introduces registers by means of logical axioms, while we have register assignment in the algebraic basis of our set-up.
8. Representing Dynamic NL Semantics

Let $D_e$ be a domain of basic entities, and assume that all registers are of type $R_{oe}$, and therefore all state changers are of the form $(r_{oe}|E_e)$. This assumption entails that the domain $D_o$ has the form $R_{oe} \rightarrow D_e$. We consider the language $L$ of dynamic predicate logic given by:

$$t := r \mid c$$

$$\phi := r \mid P^n t_1 \cdots t_n \mid (\phi_1; \phi_2) \mid \neg \phi$$

Dynamic implication $\phi_1 \Rightarrow \phi_2$ is defined as $\neg(\phi_1; \neg \phi_2)$, and the hackneyed example ‘If a farmer owns a donkey he beats it’ gets the following rendering:

$$(r_1; F r_1; r_2; D r_2; O r_1 r_2) \Rightarrow B r_1 r_2.$$

There are various ways of specifying a dynamic semantics for such a language, and these different specifications suggest different translations into TLS.

Perhaps the simplest presentation of the semantics of $L$ is as a relation $[\cdot]$ on the set $D_o$. Assume a domain of discourse $D_e$ and an interpretation $I$ for the predicates $P$ and the constants $c_e$. Let $s, s', s''$ range over $D_o$. If $s \in D_o$, then $I_s$ is the interpretation function on terms given by $I_s(c) := I(c), I_s(r) := s(r)$. The interpretation relation $[\cdot]$ for the formulas of dynamic predicate logic is given by:

$$s[r]s' \iff s' = s[r \mapsto d] \text{ for some } d \in D_e$$

$$s[P^n t_1 \cdots t_n]s' \iff s = s' \text{ and } (I_s t_1, \cdots, I_s t_n) \in I(P)$$

$$s[\phi_1; \phi_2]s' \iff \text{ there is an } s'' \text{ with } s[\phi_1]s'' \text{ and } s''[\phi_2]s'$$

$$s[\neg \phi]s' \iff s = s' \text{ and there is no } s'' \text{ with } s[\phi]s''$$

This semantic specification suggests a straightforward translation into TLS, where the translations for terms get type $(\circ, e)$ and those for dynamic formulas get type $(\circ, (\circ, t))$.

In the translation $\circ$ for terms we map registers to themselves, ordinary constants to constant functions in $(\circ, e)$:

$$r^\circ := r$$

$$c^\circ := \lambda i. c$$

The translation $\bullet$ for formulas is a completely straightforward rendering of the semantic specifications in TLS:

$$r^\bullet := \lambda i j. \exists x_e((r|x_e)i = j)$$

$$(P^n t_1 \cdots t_n)^\bullet := \lambda i j. (i = j \land P(t^n_1 i) \cdots (t^n_n i))$$

$$(\phi_1; \phi_2)^\bullet := \lambda i j. ((\phi_1^\bullet i k \land \phi_2^\bullet k j))$$

$$(\neg \phi)^\bullet := \lambda i j. (i = j \land \neg \exists k \phi^\bullet i k)$$

This is essentially the translation given in [21].

The dynamic semantics above can also, equivalently, be given in functional style, as a mapping from sets of states to sets of states, as follows:

$$[r](A) := \{ s[r \mapsto d] \mid s \in A, d \in D_e \}$$
\[ \llbracket P^n t_1 \cdots t_n \rrbracket (A) := \{ s \in A \mid (I_s t_1, \ldots, I_s t_n) \in I(P) \} \]
\[ \llbracket \phi_1; \phi_2 \rrbracket (A) := [\llbracket \phi_2 \rrbracket ([\llbracket \phi_1 \rrbracket (A)]) \]
\[ [\neg \phi] (A) := \{ s \in A \mid [\phi] (s) = \emptyset \} \]

This suggests a translation in type \((\langle \circ, t \rangle \langle \circ, t \rangle)\), as follows (assume \(p\) is a variable of type \((\circ, t)\)):

\[
\begin{align*}
r^\circ & := \lambda p. \lambda i. \exists x (p j \land (r | x) j = i) \\
(P^n t_1 \cdots t_n)^\circ & := \lambda p. \lambda i. (p t_i \land P(t_1^i) \cdots (t_n^i)) \\
(\phi_1; \phi_2)^\circ & := \lambda p. \phi_2^\circ (\phi_1^p p) \\
(\neg \phi)^\circ & := \lambda p. \lambda i. (p t_i \land \neg \exists k \phi^\circ (\lambda j. j = i) k)
\end{align*}
\]

Less obviously, it is also possible to specify the semantics for \(L\) as a function mapping states to sets of sets of states:

\[ [\phi](s) := \{ A \subseteq D_0 \mid s' \in A \text{ for some } s' \text{ with } s[\phi]s' \}. \]

This is the basis for the system of Dynamic Montague Grammar proposed in \textsuperscript{[10]}. Essentially, the DMG translation of \(L\) boils down to:

\[ \phi^\dagger := \lambda p. \exists j (\phi^* ij \land pj). \]

Note that this translation has \(i\) free. In fact, the DMG translation uses Montague’s intensional typed logic with \(\cup\) and \(\cap\) (IL), but every IL formula has a corresponding typed logic formula in a language with one extra basic type (in our case: \(\circ\), with possibly one extra variable \(i\) free in this basic type (see Gallin \textsuperscript{[9]}).

In our framework, it would be more natural to abstract over the state variable. This yields the following translation into type \((\langle \circ, ((\circ, t), t) \rangle)\):

\[ \phi^\dagger := \lambda i. \lambda p. \exists j (\phi^* ij \land pj). \]

To see that this higher type does make sense, let us bother to spell out the semantics for \(L\) as a function in \(D_0 \rightarrow P P D_0\).  

\[
\begin{align*}
[\| r \| (s) & := \{ A \subseteq D_0 \mid s[r \mapsto d] \in A \text{ for some } d \in D_0 \} \\
[\| P^n t_1 \cdots t_n \| (s) & := \{ A \subseteq D_0 \mid s \in A \text{ and } (I_s t_1, \ldots, I_s t_n) \in I(P) \} \\
[\| \phi_1; \phi_2 \| (s) & := \bigcup \{ [\| \phi_2 \| (s') \mid \{ s' \} \in [\| \phi_1 \| (s)] \} \\
[\| \neg \phi \| (s) & := \{ A \subseteq D_0 \mid s \in A \text{ and } [\| \phi \| (s) = \{ \emptyset \} \}
\end{align*}
\]

Because of the equation

\[ [\| \phi \| (s) = \{ A \subseteq D_0 \mid s' \in A \text{ for some } s' \text{ with } s[\phi]s' \} \]

we have that sets in \([\| \phi \| (s)\) are always up-sets, in the sense that they are all of the form \(\uparrow B\), where \(\uparrow B := \{ B' \supseteq B \mid B \in B \}. \) Indeed, we can write the function \([\| \cdot \|\) as follows:

\[ [\| \phi \| (s) = \uparrow \{ s' \mid s[\phi]s' \}. \]
In [10] an alternative negation $\sim$ is proposed, to be used in addition to $\neg$, with semantics

$$\llbracket \sim \phi \rrbracket(s) := \mathcal{P}\mathcal{D}_{s} - \llbracket \phi \rrbracket(s).$$

For an extensive discussion of the pros and cons of this dynamic negation see the second chapter of Dekker [4]. In brief, the meaning of an expression containing $\sim$ negations will not be an up-set, for if $\llbracket \phi \rrbracket(s)$ is an up-set, $\llbracket \sim \phi \rrbracket(s)$ is a down-set, and vice versa. This is slightly problematic, for in general one wants that the meaning of an expression acts as an information update, and information updates correspond to up-sets. On the positive side, we have that $\llbracket \phi \rrbracket(s)$ and $\llbracket \sim \phi \rrbracket(s)$ are the same, so we recover the law of double negation at the dynamic level.

Note that $\sim \phi$ is also readily translated into TLS, by means of the following extension of $\dagger$:

$$(\sim \phi)^\dagger := \lambda i \lambda p. (\neg ((\phi^\dagger p)i)).$$

The further extensions proposed in Dekker [4] also have obvious translations in TLS.

9 Comparisons

In this section we will compare TLS with some of the typed logics that have been proposed for integrating discourse representation theory [16, 7] into a Montague-style compositional framework [3, 10, 20, 21, 18]. The proposals that come closest to the present approach are [3] and [20].

The two main flavours of dynamic typed logic are a dynamic version of Montague’s IL, and a dynamic version of Gallin’s Ty2. Both of these take a typed logic in which possible worlds either figure as a domain for forming intensions (IL) or as a basic type (Ty2), and impose extra meaning postulates intended to force these possible worlds to behave like storage states for a given set of store constants.

One good reason for allowing $\lambda$ abstraction over states, i.e., for following the Ty2 tradition rather than the IL route, is that an extension with ‘real’ possible worlds is trivial. Just build your TLS models over a basic type set $B$ which includes the type $s$, and interpret $s$ as the domain $D_s$ of possible worlds. Translations of dynamic modal predicate logic [5] and similar logics into TLS are now straightforward. Also, TLS comes with a notion of reduction which is both precise and elegant, while reduction of expressions in dynamic versions of IL is hopelessly cumbersome.

The standard procedure for distilling a typed logic with states out of ordinary typed logic is as follows (we assume for simplicity that the values to be stored all have the same type $T$): in the footsteps of Janssen [15], one starts out with an arbitrary domain $D_s$, one introduces a set $C$ of ‘store constants’ of type $(s, T)$ (where $T$ does not depend on $s$), and one tries to force $D_s$ to behave as the function space $C \to D_T$. The two principles needed to carry out this procedure are:

1. states can only differ in the values of the store constants,
2. there are enough states, in the sense that an update of an arbitrary store constant with an arbitrary value will always yield a new state.

Unfortunately, it is impossible to formulate a postulate for (1) as a statement of the typed language. A postulate for (1) must refer to a constant REG which is assumed...
to be true of a term $E$ iff $E$ is a register of values of type $T$. Assume for simplicity that all registers store values of the same type $T$, and consider the following postulate.

$$\text{DIST} \quad \forall i, j_s (\forall x_{(s,T)} (\text{REG } x \rightarrow (xi = xj)) \rightarrow i = j).$$

Obviously, given a compositional semantics, there can be no interpretation function with $\llbracket \text{REG } x \rrbracket = 1$ iff $x$ itself is a register for values of type $T$, simply because compositionality means that the objects in a model (in this case: the object $\llbracket x \rrbracket$) do not reveal the names that have been used to refer to them (in this case: the register name $x$).

Note, however, that what DIST attempts to express is true for all TLS models. The following principle (which cannot be expressed as a formula of the language for reasons explained above) holds for every general model $M = (\{ D_U \}, A_o, \{ A_{UU'} \})$ and every assignment $g$ for $M$:

$$[i = j]^M_g = 1 \quad \text{iff} \quad g(i) = g(j)$$
$$\text{iff} \quad A_o(g(i), r) = A_o(g(j), r) \text{ for all } r \in R.$$

This illustrates that distributivity is built into the TLS general models (and therefore also into the full models) from the start.

Since (1) cannot be enforced by a formula of standard typed logic, one way around it is to make a virtue out of a necessity by dropping the requirement that states are only made up out of the named registers. This is what Muskens proposes in [20]. Here a constant $\text{ST}$ is introduced to be interpreted as the property of being a store function. Appropriate axioms are employed to ensure that store functions are the stuff that states are made of. Assume that $\text{ST}$ is a constant for the property of being a store function. Then the formula

$$\forall x_{(s,T)} ((\text{ST } x \land x \neq c) \rightarrow (xi = xj))$$

can abbreviate the relation of differing at most in the value of store register $c$. Notation: $i[c]j$. Next, it is ensured by means of axioms that there are enough states and that at least all store constants are interpreted as store functions, as follows:

$$\text{AX 1} \quad \forall i \forall x \forall e (\text{ST } x \rightarrow \exists j ([i x] j \land xj = e))$$
$$\text{AX 2} \quad \text{ST } c \text{ for each store constant } c$$
$$\text{AX 3} \quad c \neq c' \text{ for each pair of different store constants } c, c'.$$

To see that these axioms are consistent, consider the following model (personal communication from Reinhard Muskens): states are the functions $s \in \mathbb{N} \rightarrow \mathbb{N}$ with the property that $\{ n \in \mathbb{N} \mid s(n) \neq 0 \}$ is finite. This set of states is denumerable, so we can arrange states and registers in a square as follows:

<table>
<thead>
<tr>
<th></th>
<th>$u_1$</th>
<th>$u_2$</th>
<th>$u_3$</th>
<th>$u_4$</th>
<th>$u_5$</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_2$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Now the columns can be taken to be the interpretations of the register constants, so \(u_i\) is interpreted as the \(i\)-th column. It is not difficult to see that the store functions are precisely those \(F \in S \to \mathbb{N}\) with the property that there is some \(n \in \mathbb{N}\) such that for all \(s \in S\), \(F(s) = s(n)\). \((n\) is the index of the column\) Therefore we put \(STf\) true if \(f\) corresponds to a column.

One thing to check now is that the store functions are indeed independent. We want to ensure that, for example, \(\lambda i. u_1 i + u_2 i\), does not correspond to a register. But this is easy. Take an arbitrary register (column) \(u_i\), and an arbitrary state (row) \(s_j\). Suppose we put a new value \(m\) at the position of \(s_j(i)\) (the place where the column and the row intersect). Then the resulting row must be present somewhere in the table, for it is again a function \(s\) with the property that \(\{ n \in \mathbb{N} | s(n) \neq 0 \}\) is finite. But this means that \(u_i\) must be independent of the other stores. Thus, \(\lambda i. u_1 i + u_2 i\) does not correspond to a store.

Note that all the axioms are true in this model. This shows that the axioms are consistent. Note that the axioms do not ensure that the states are built of precisely the registers mentioned in the axioms. For an example of a model with anonymous registers, consider the following slight extension of the previous model: states and interpretation of \(ST\) as before, interpretation of registers as follows (\(A_1, A_2, A_3, \ldots\) are anonymous registers):

\[
\begin{array}{cccccc}
  u_1 & A_1 & u_2 & A_2 & u_3 & \cdots \\
  s_1 \\
  s_2 \\
  s_3 \\
  s_4 \\
  s_5 \\
  \vdots
\end{array}
\]

This is also a model of the axioms AX1–AX3. It seems that the choice between the approach of Muskens and the present approach depends on whether one wants DIST or not.

The postulate that should guarantee requirement (2) (enough states) can be formulated in TLS as follows:

\[
\text{UPDATE} \quad \forall i \forall x \exists j (j = (c|x)i)
\]

We have already seen above that the UPDATE postulate is true in all TLS models (note that in Muskens’ approach UPDATE is guaranteed by AX 1).

10 Further Work

The two main areas of further work are:

- exploring further use of TLS as a tool,
- further logical investigation of TLS.

Proposals for dynamic semantics are more readily compared when they are formulated within the same framework. In the area of NL semantics, suitable candidates for translation into TLS are the sequence semantics for dynamic predicate logic from
Typed Logics With States

the extension of dynamic predicate logic with procedures from 15, the modalized versions of dynamic predicate logic of 16, 12, and so on.

In the area of programming language semantics, static program analysis for while programs is easily performed within TLS. Note, for instance, that partial correctness assertions \( \{P\} S \{Q\} \) are readily translated into TLS formulas by means of:

\[
\forall ij ((P_i \land S^* ij) \rightarrow Q_j),
\]

while total correctness assertions take the form:

\[
\forall i (P_i \rightarrow \exists j (S^* ij \land Q_j)).
\]

In a slightly different direction, TLS is a suitable tool for variable dependency analysis of while programs, including reasoning about safety of the analysis with respect to a given semantic specification (see 23 for examples of such reasoning).

There are further logical questions to be asked about TLS. Modifications of the \( \sigma \), \( \rho \) and \( \tau \) axioms may provide an interesting connection with Van Benthem’s weak predicate logics 27, which are also the result of varying the restrictions on the set of available variable assignments. As an example, consider the \( \rho \) axiom permitting the swap between \((r_i|E)(r_j|F)G\) and \((r_j|F)(r_i|E)G\). This expresses independence of the registers; if we drop this axiom we allow models with states where register \( r_i \) may depend in some way on \( r_j \) or vice versa. In a similar way, one can consider dropping the other \( \rho \) axiom, \((r_i|E)(r_i|F)G = (r_i|E)G\), which expresses the destructiveness of register update. If one considers \((r_i|E)(r_i|F)G\) and \((r_i|E)G\) as different states, this means that registers effectively become stacks. Together with an appropriate mechanism for register lookup (lookup at arbitrary depths in the register stacks, or an operator for throwing the top of the stack away) this would give us the typing superstructure for a dynamic logic with stack updates in the spirit of the already mentioned 27.

Acknowledgements

Thanks to Erik Barendsen, Johan van Benthem, Paul Dekker, Theo Janssen, Jan-Willem Klop, Reinhard Muskens, Femke van Raamsdonk and Anne Troelstra for helpful discussions, and to four anonymous referees of this Journal for spurring me on to considerable improvements with their feedback on previous versions of this paper. All remaining errors are my own.

References

10. FURTHER WORK


Received 23 October 1996. Revised 26 June 1997