```
/*
Copyright 2013 Jochem H. Rutgers (j.h.rutgers@utwente.nl)

This file is part of lambda.

lambda is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

lambda is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with lambda.  If not, see <http://www.gnu.org/licenses/>.
*/

#ifndef __LAMBDA_PTR_H
#define __LAMBDA_PTR_H

////////////////////////////////////
////////////////////////////////////
// Pointer-on-stack tracing for GC
////////////////////////////////////
////////////////////////////////////

#include <lambda/debug.h>

namespace lambda {

    ////////////////////////////////////
    // Pointer stack

    class Term;
    template <typename T> class T_tptr;

    template <typename T,bool enabled> struct enable_type { typedef T type; };
    template <typename T> struct enable_type<T,false> { typedef T type[0]; };

//  static __thread T_tptr<Term>* term_stack;
    DECL_THREAD_LOCAL_PTR(T_tptr<Term>*,term_stack)
#define term_stack DECL_THREAD_LOCAL_PTR_NAME(term_stack,0)

    template <typename T,Config::gc_type_t gc=Config::gc_type>
    class T_stackable {
    public:
        T_stackable() LAMBDA_INLINE : m_prev(NULL) {}
        ~T_stackable() LAMBDA_INLINE {
            if(likely(peek()!=NULL)){
                LAMBDA_PRINT(refs,"ptr %p -> %p destructed, pop from stack, new top is
                %p",that(),that()->ptr(),peek());
                pop();
            }else
                LAMBDA_PRINT(refs,"temporary ptr %p -> %p destructed",that(),that()->ptr());
        }
        T_tptr<T>* peek() const LAMBDA_INLINE {
            return m_prev;
        }
//  protected:
        void push() const LAMBDA_INLINE {
            LAMBDA_ASSERT(m_prev==NULL,"doubly stacked ptr %p -> %p",that(),that()->ptr());
//          LAMBDA_PRINT(refs,"push %p -> %p, prev is %p",that(),that()->ptr(),term_stack);
            m_prev=term_stack;
            term_stack=that();
        }
```

```
        void pop() const LAMBDA_INLINE {
            if(term_stack==that()){
//              LAMBDA_PRINT(refs,"pop %p from stack, new top is %p",that(),peek());
                term_stack=peek();
                m_prev=NULL;
            }else{
                T_stackable* top=static_cast<T_stackable*>(term_stack);
                LAMBDA_ASSERT(top->peek()==that(),
                    "expected stack top to be: (top) %p; (this) %p, found (top) %p;
                    %p",term_stack,that(),term_stack,top->peek());
//              LAMBDA_PRINT(refs,"swap and pop %p from stack, top is %p",that(),term_stack);
                top->m_prev=peek();
                m_prev=NULL;
            }
        };
        void swap(T_tptr<T>* p) const LAMBDA_INLINE {
            LAMBDA_ASSERT(peek()!=NULL,"cannot swap non-stacked %p -> %p by %p",that(),that()-
            >ptr(),p);
            p->m_prev=m_prev;
            m_prev=p;
        }
    private:
        T_tptr<T>* that() LAMBDA_INLINE { return static_cast<T_tptr<T>*>(this); }
        T_tptr<T>* that() const LAMBDA_INLINE { return
        const_cast<T_tptr<T>*>(static_cast<const T_tptr<T>*>(this)); }
        mutable T_tptr<T>* m_prev;
    };

    template <typename T>
    class T_stackable<T,Config::gc_none> {
    public:
        static T_tptr<T>* peek()        LAMBDA_INLINE { return NULL; }
    protected:
        static void push()              LAMBDA_INLINE {}
        static void pop()               LAMBDA_INLINE {}
        static void swap(T_tptr<T>* p)  LAMBDA_INLINE {}
    };

    ////////////////////////////////
    // Pointer type

    template <typename T> class T_tref;

    template <typename T>
    class T_tptr : public T_stackable<T,Config::gc_type> {
    public:
        typedef T_stackable<T,Config::gc_type> base;

        T_tptr(T* t=NULL) LAMBDA_INLINE : base(), m_t(t) {
            this->push();
            LAMBDA_PRINT(refs,"new temporary ptr %p -> %p, stack was %p",this,m_t,this-
            >peek());
        }
        T_tptr(const T_tptr& r) LAMBDA_INLINE : base(), m_t(r.ptr()) {
            this->push();
            LAMBDA_PRINT(refs,"new temporary ptr %p by copy of (temporary) ptr %p -> %p, stack
            was %p",this,&r,r.ptr(),this->peek());
        }
        T_tptr(const T_tref<T>& r) LAMBDA_INLINE : base(), m_t(r.ptr()) {
            this->push();
            LAMBDA_PRINT(refs,"new temporary ptr %p by copy of (temporary) ref %p -> %p
            (swapped)",this,&r,r.ptr());
        }
        ~T_tptr() LAMBDA_INLINE {}
        T& operator*() const LAMBDA_INLINE {
            return *ptr();
        }
```

```
    T* operator->() const LAMBDA_INLINE {
        return &operator*();
    }
    operator T*() const LAMBDA_INLINE {
        return &operator*();
    }
    bool operator==(T* rhs) LAMBDA_INLINE {return ptr()==rhs;}
    bool operator==(const T_tptr& rhs) LAMBDA_INLINE {return ptr()==rhs.ptr();}
    bool operator!=(T* rhs) LAMBDA_INLINE {return ptr()!=rhs;}
    bool operator!=(const T_tptr& rhs) LAMBDA_INLINE {return ptr()!=rhs.ptr();}
    T* ptr() const LAMBDA_INLINE {
        return m_t;
    }
protected:
    T_tptr(T* t,bool do_push) LAMBDA_INLINE : base(), m_t(t) {if(do_push)this->push();}
    // this must be a temporary!
    T_tptr& operator=(const T_tptr& rhs);
    void SetPtr(T* t) LAMBDA_INLINE { m_t=t; }
private:
    T* m_t;
};

template <typename T>
class T_ptr : public T_tptr<T> {
public:
    typedef T_tptr<T> base;
    T_ptr(T* t=NULL) LAMBDA_INLINE : base(t,true) {
        LAMBDA_PRINT(refs,"new ptr %p -> %p, stack was %p",this,this->ptr(),this->peek());
    }
    T_ptr(const T_ptr& p) LAMBDA_INLINE : base(p.ptr(),true) {
        LAMBDA_PRINT(refs,"new ptr to ptr %p -> %p -> %p, stack was
        %p",this,&p,p.ptr(),this->peek());
    }
    T_ptr(const T_tptr<T>& p) LAMBDA_INLINE : base(p.ptr(),false) {
        LAMBDA_PRINT(refs,"new ptr %p from temporary %p -> %p (swapped)",this,&p,this-
        >ptr());
        p.swap(this);
    }
    T_ptr(const T_tref<T>& r) LAMBDA_INLINE : base(r.ptr(),true) {
        LAMBDA_PRINT(refs,"new ptr %p by copy of (temporary) ref %p -> %p, stack was
        %p",this,&r,r.ptr(),this->peek());
    }
    ~T_ptr() LAMBDA_INLINE {}
    T_ptr& operator=(const T_ptr& rhs) LAMBDA_INLINE {
        LAMBDA_PRINT(refs,"set ptr %p to %p by ptr %p",this,rhs.ptr(),&rhs);
        return operator=(rhs.ptr());
    }
    T_ptr& operator=(const T_tptr<T>& rhs) LAMBDA_INLINE {
        LAMBDA_PRINT(refs,"set ptr %p to %p by temporary %p",this,rhs.ptr(),&rhs);
        return operator=(rhs.ptr());
    }
    T_ptr& operator=(T* rhs) LAMBDA_INLINE {
        this->SetPtr(rhs);
        return *this;
    }
};

////////////////////////////////////
// Reference type

template <typename T> class T_ref;

template <typename T>
class T_tref : protected T_tptr<T> {
public:
    typedef T_tptr<T> base;
    explicit T_tref(T& t) LAMBDA_INLINE : base(&t,true) {
```

```
            LAMBDA_PRINT(refs,"new temporary ref %p -> %p, stack was %p",this,this->ptr(),this->peek());
        }
        T_tref(const T_tref& r) LAMBDA_INLINE : base(r.ptr(),false) {
//          LAMBDA_ASSERT(r.peek()==NULL,"cannot copy temporary ref %p -> %p that is on the stack",&r,r.ptr());
            LAMBDA_PRINT(refs,"new temporary ref %p by copy of temporary ref %p -> %p (swapped)",this,&r,r.ptr());
            r.swap(this);
        }
        T_tref(const T_ref<T>& r) LAMBDA_INLINE : base(r.ptr(),true) {
//          LAMBDA_ASSERT(r.peek()==NULL,"cannot copy temporary ref %p -> %p that is on the stack",&r,r.ptr());
            LAMBDA_PRINT(refs,"new temporary ref %p by copy of ref %p -> %p, stack was %p",this,&r,r.ptr(),this->peek());
        }
        T_tref(const T_tptr<T>& p) LAMBDA_INLINE : base(p.ptr(),false) {
//          LAMBDA_ASSERT(p.peek()==NULL,"cannot copy temporary ptr %p -> %p that is on the stack",&p,p.ptr());
            LAMBDA_PRINT(refs,"new temporary ref %p by copy of temporary ptr %p -> %p (swapped)",this,&p,p.ptr());
            p.swap(this);
        }
        T_tref(const T_ptr<T>& p) LAMBDA_INLINE : base(p.ptr(),true) {
//          LAMBDA_ASSERT(p.peek()==NULL,"cannot copy temporary ptr %p -> %p that is on the stack",&p,p.ptr());
            LAMBDA_PRINT(refs,"new temporary ref %p by copy of ptr %p -> %p, stack was %p",this,&p,p.ptr(),this->peek());
        }
        ~T_tref() LAMBDA_INLINE {}
        operator T&() const LAMBDA_INLINE {
            return term();
        }
        T& term() const LAMBDA_INLINE {
            LAMBDA_ASSERT(ptr()!=NULL,"cannot have null-ref by %p\n",this);
            return base::operator*();
        }
        T* ptr() const LAMBDA_INLINE {
            return base::ptr();
        }
        T_tptr<T>* peek() const LAMBDA_INLINE {
            return base::peek();
        }
        void push() const LAMBDA_INLINE {
            return base::push();
        }
        void pop() const LAMBDA_INLINE {
            return base::pop();
        }
        void swap(T_tref<T>* r) const LAMBDA_INLINE {
            return base::swap(r);
        }
        T* operator&() LAMBDA_INLINE {
            return &(T&)*this;
        }

        // be a wrapper for T=Term
        T_tref Apply(T& a)                          LAMBDA_INLINE { return term().Apply(a); }
        T_tref operator()(T& t)                     LAMBDA_INLINE { return term().operator()(t); }
//      T_tref operator()(lcint_t c)                LAMBDA_INLINE { return term().operator()(c); }
        T_tref operator()(int c)                    LAMBDA_INLINE { return term().operator()(c); }
        T_tref operator()(long c)                   LAMBDA_INLINE { return term().operator()(c); }
        T_tref operator()(long long c)              LAMBDA_INLINE { return term().operator()
```

```
        (c); }
        T_tref operator()(lcfloat_t c)              LAMBDA_INLINE { return term().operator()
        (c); }
        T_tref operator()(lccomplex_t c)            LAMBDA_INLINE { return term().operator()
        (c); }
        T_tref operator()(const T_tref& r)          LAMBDA_INLINE { return term().operator()
        (r); }
        T_tref operator+(T& rhs)                    LAMBDA_INLINE { return
        term().operator+(rhs);}
        T_tref operator-(T& rhs)                    LAMBDA_INLINE { return
        term().operator-(rhs);}
        T_tref operator*(T& rhs)                    LAMBDA_INLINE { return
        term().operator*(rhs);}
        T_tref operator/(T& rhs)                    LAMBDA_INLINE { return
        term().operator/(rhs);}
        int Arguments()                             LAMBDA_INLINE { return
        term().Arguments();}
        T_tref<T> Reduce()                          LAMBDA_INLINE { return term().Reduce();}
        T_tref<T> ReduceApply(T* a1=NULL,T* a2=NULL,T* a3=NULL,T* a4=NULL,T* a5=NULL)
                                                    LAMBDA_INLINE { return
                                                    term().ReduceApply(a1,a2,a3,a4,a5);}
        T_tref<T> FullReduce()                      LAMBDA_INLINE { return
        term().FullReduce();}
//      bool operator==(T& rhs)                     LAMBDA_INLINE { return
term().operator==(rhs);}
        const void* Compute()                       LAMBDA_INLINE { return term().Compute();}
        template <typename R> const R& LAMBDA_INLINE Compute()    { return
        term().Compute<R>();}
    protected:
        T_tref(T& t,bool do_push) LAMBDA_INLINE : base(&t,do_push) {}
        // this must be a temporary!
        T_tref& operator=(const T_tref& rhs);
    };

    template <typename T>
    class T_ref : public T_tref<T> {
    public:
        typedef T_tref<T> base;
        explicit T_ref(T& t) LAMBDA_INLINE : base(t,true) {
            LAMBDA_PRINT(refs,"new ref %p -> %p, stack was %p",this,&t,this->peek());
        }
        T_ref(const T_ref& r) LAMBDA_INLINE : base(*r.ptr(),false) {
            LAMBDA_PRINT(refs,"new ref to ref %p -> %p -> %p, not on stack",this,&r,r.ptr());
        }
        T_ref(const T_tref<T>& r) LAMBDA_INLINE : base(*r.ptr(),false) {
            LAMBDA_PRINT(refs,"new ref %p from temporary ref %p -> %p
            (swapped)",this,&r,r.ptr());
            r.swap(this);
        }
        T_ref(const T_tptr<T>& p) LAMBDA_INLINE : base(*p.ptr(),false) {
            LAMBDA_PRINT(refs,"new ref %p -> ptr %p -> %p (swapped)",this,&p,p.ptr());
            p.swap(this);
        }
        ~T_ref() LAMBDA_INLINE {}
    protected:
        // ref is single-assignment
        T_ref& operator=(const T_ref& rhs);
    };

    ////////////////////////////////
    // Term types

    typedef T_tptr<Term> Term_tptr;
    typedef T_ptr<Term> Term_ptr;
    typedef T_tref<Term> Term_tref;
    typedef T_ref<Term> Term_ref;
    typedef Term_ref let;
```

```
};

#endif // _LAMBDA_PTR_H
```